pascal

MANUALE E STANDARD DEL LINGUAGGIO

EDIZIONE ITALIANA

Kathleen Jensen
Niklaus Wirth

GRUPPO EDITORIALE JACKSON

pascal

pascal

MANUALE E STANDARD DEL LINGUAGGIO

di
Kathleen Jensen
Niklaus Wirth

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano
Tel. 680368 - 680054 - 68951/2/3/4/5

© Copyright per l'edizione originale Springer-Verlag New York Inc. 1975

© Copyright per l'edizione italiana Gruppo Editoriale Jackson 1981

Il Gruppo Editoriale Jackson ringrazia per il prezioso lavoro svolto nella stesura dell'edizione italiana le signore Francesca di Fiore, Rosi Bozzolo e l'Ing. Roberto Pancaldi.

Tutti i diritti sono riservati. Stampato in Italia. Nessuna parte di questo libro può essere riprodotta, memorizzata in sistemi di archivio, o trasmessa in qualsiasi forma o mezzo, elettronico, meccanico, fotocopia, registrazione o altri senza la preventiva autorizzazione scritta dell'editore.

Stampato in Italia da:
S.p.A. Alberto Matarelli - Milano - Stabilimento Grafico

PREFAZIONE

Una versione preliminare del linguaggio di programmazione Pascal è stata stesa nel 1968. Derivava, nella sostanza, dai linguaggi Algol-60 e Algol-W. Dopo un'ampia fase di sviluppo, un primo compilatore divenne operativo nel '70 mentre un anno più tardi seguirono alcune pubblicazioni (si vedono i riferimenti 1 e 8 a pag. 113). Se da un lato il crescente interesse nello sviluppo di compilatori per altri elaboratori richiese il consolidamento di Pascal, dall'altro due anni di esperienza nell'uso del linguaggio imposero alcune revisioni. Questo portò a pubblicare, nel '73, una revisione del rapporto e la definizione di una rappresentazione del linguaggio secondo l'insieme dei caratteri ISO.

Questo libro è costituito da due parti: il manuale utente ed il rapporto rivisto. Il *manuale* è orientato verso coloro i quali hanno già acquistato una certa familiarità con la programmazione e che vogliono essere informati sul linguaggio Pascal. Lo stile ed i molti esempi, inclusi per mostrare le varie caratteristiche di Pascal, danno al manuale un carattere didattico. Tavole riassuntive e specifiche sintattiche sono riportate nelle appendici. Il *Rapporto* è incluso in questo libro per servire come conciso e definitivo riferimento sia per i programmatori che gli implementatori del compilatore. Definisce un *Pascal standard* che costituisce la base comune per le diverse implementazioni del linguaggio.

La struttura lineare di un libro non è certo l'ideale per introdurre un linguaggio. Ciò nonostante, nell'uso didattico, raccomandiamo di seguire l'organizzazione data al manuale, prestando molta attenzione ai programmi degli esempi; converrà in seguito rivedere i paragrafi che hanno destato particolari difficoltà. In particolare, se sorgono domande sulle convenzioni di ingresso ed uscita, si deve fare riferimento al capitolo 12.

I capitoli 0-12 del manuale, e l'intero rapporto, descrivono Pascal standard. Un implementatore dovrebbe considerare il compito di riconoscere Pascal standard come la richiesta base del suo sistema, mentre il programmatore, il quale desideri che i suoi programmi siano trasferibili da un elaboratore all'altro, dovrebbe usare solo le caratteristiche descritte come Pascal standard. Naturalmente, certe implementazioni possono fornire prestazioni addizionali che, tuttavia, dovrebbero essere chiaramente indicate come estensioni.

I capitoli 13 e 14 del manuale documentano l'implementazione di Pascal sull'elaboratore Control data 6000. Il capitolo 13 descrive le caratteristiche addizionali del linguaggio chiamato Pascal 6000/3.4. Il capitolo 14 spiega l'uso del compilatore sotto il sistema operativo SCOPE 3.4.

Il manuale è frutto dello sforzo di parecchie persone; un ringraziamento particolare dobbiamo ai membri dell'Istituto di informatica, ETH ZURIGO, a Gohn Larmouth, Rudy Schild, Olivier Lecarme, e Pierre Desjardins per le critiche, i suggerimenti e l'incoraggiamento. La nostra implemetazione di Pascal, che rende questo manuale possibile e necessario, è lavoro di Urs Ammann aiutato da Helmut Sandmayr.

Kathleen Jensen
Niklaus Wirth
ETH Zurigo
Svizzera

INTRODUZIONE ALL'EDIZIONE ITALIANA

A dieci anni dalla "nascita" di Pascal si può sicuramente affermare che lo sforzo di rompere il circolo vizioso tra "la scelta di un linguaggio per insegnare basato sulla sua diffusione ed il fatto che il linguaggio più insegnato diviene il più diffuso" è stato coronato da successo. L'aver rotto quel cerchio non solo ha permesso di rendere l'insegnamento della programmazione "una disciplina sistematica basata su certi concetti riflessi in modo chiaro e naturale dal linguaggio" ma ha creato un mondo Pascal.
Questo mondo è stato arricchito dai contributi di molti studiosi ed ampliato con la scrittura di nuovi compilatori da parte di parecchie aziende. Uno dei contributi più significativi è certamente stato dato dall'università di S. Diego (California) con il suo Pascal UCSD. Delle due classi di compilatori, generanti l'una un codice intermedio (P Code) e l'altra un codice nativo, UCSD ha scelto la prima. Questo ha permesso di sviluppare un pacchetto, basato su Pascal, che gira su un buon numero di micro e mini-elaboratori. Altre industrie, privilegiando, nel compromesso efficienza-portabilità, il primo polo hanno progettato dei compilatori appartenenti alla seconda classe, il che permette loro di abbandonare l'assembler in favore di Pascal nello sviluppo del software di base. Al di là dei contributi diretti al linguaggio, l'atmosfera del "pensare in Pascal" ha, comunque, talmente influenzato il campo dell'informatica che non nasce un linguaggio che non debba confrontarsi con Pascal; confrontandosi: se non addirittura essere considerato un sottoinsieme od un ampliamento dello stesso. È certamente in questo mondo che stanno e nasceranno i linguaggi del domani.
Ma oltre cortina, nel vecchio mondo, i "coboliani" da buon senso contadino continuano a darsi virili botte sulle spalle mentre esaltano la diffusione, la chiarezza e la flessibilità del COBOL e additano, allo schivo sorriso dei FORTRAN-men, la dotta prosopopea dei sostenitori di Pascal. È quasi spiacevole rompere quel'incanto da "asinus asinum fricat" ma vorremmo far notare che usando concetti ben definiti come: produttività, manutenibilità, affidabilità, leggibilità... il confronto, qualunque sia il capo di applicazione, è perdente per il vecchio mondo. Naturalmente se il confronto è fatto correttamente, tenendo cioè presente l'ambiente in cui il programma è inserito (sono le caratteristiche del sistema operativo che permettono accessi più o meno sofisticati ai flussi).
Con queste premesse la domanda se il testo di Hirth dovesse venir tradotto diviene retorica. In particolare, per il lettore italiano, abituato a testi-fiume, esiste anche un motivo didattico: un testo essenziale e sintetico, ricco di esempi, e povero di giri di parole vuote forza il lettore, quasi ad ogni riga, a verificare il suo livello di comprensione senza illudersi con facili e fumosi "ho capito".

Ivan Maffezzini

SOMMARIO

Prefazione III

Introduzione 1

CAPITOLO 1 - NOTAZIONE E VOCABOLARIO 7

CAPITOLO 2 - IL CONCETTO DI DATO 11
A. Il tipo booleano (boolean) 11
B. Il tipo intero (integer) 12
C. Il tipo reale (real) 13
D. Il tipo carattere (char) 14

CAPITOLO 3 - INTESTAZIONE DEL PROGRAMMA E SEZIONE DI DICHIARAZIONE 17
A. Intestazione programma 17
B. Parte dichiarazione etichette 17
C. Parte definizione costanti 18
D. Parte definizione tipo 19
E. Parte dichiarazione variabili 19
F. Parte dichiarazione procedure e funzioni 20

CAPITOLO 4 - IL CONCETTO DI AZIONE 21
A. Istruzione di assegnamento 21
B. Istruzione composta 23
C. Istruzione di iterazione 23
C.1 L'istruzione while 23
C.2 L'istruzione repeat 24
C.3 L'istruzione for 25
D. Istruzioni condizionali 27
D.1 L'istruzione if 27
D.2 L'istruzione case 32
E. L'istruzione di goto 32

CAPITOLO 5 - TIPI SCALARE E SOTTOCAMPI 35
A. Tipi scalare 35
B. Tipi sottocampo 36

CAPITOLO 6 - GENERALITA' SUI TIPI STRUTTURATI - I VETTORI IN PARTICOLARE 39

CAPITOLO 7 - TIPI RECORD 45
A. L'istruzione with 51

CAPITOLO 8 - TIPI SET 55

CAPITOL 9 - TIPI FLUSSO (FILE) 61
A. Flussi-testo 64
B. I flussi standard "input" e "output" 65

CAPITOLO 10 - TIPI PUNTATORE 69

CAPITOLO 11 - PROCEDURE E FUNZIONI 75
A. Procedure 75
B. Funzioni 86
C. Note 90

CAPITOLO 12 - INGRESSO E USCITA 93
A. Procedura read 94
B. Procedura write 95

CAPITOLO 13 - PASCAL 6000-3.4 97
A. Estensioni al linguaggio Pascal 97
A.1 Flussi segmentati 97
A.2 Procedure esterne 100
B. Specifiche lasciate indefinite nei capitoli precedenti 100
B.1 Intestazione dei programmi e flusso esterni 100
B.2 Rappresentazione dei flussi 101
B.3 I tipi standard 102
B.4 La procedura standard "write" 105
C. Restrizioni 106
D. Tipi, procedure e funzioni predefiniti aggiunti 107
D.1 Tipi predefiniti aggiunti 107
D.2 Funzioni e procedure predefinite aggiunte 107

CAPITOLO 14 - Come usare il sistema PASCAL 6000-3.4 109
A. Direttive di controllo 109
B. Opzioni del compilatore 110
C. Messaggi di errore 112
C.1 Compilatore 112
C.2 In esecuzione 112

RIFERIMENTI 113

APPENDICE A PROCEDURE E FUNZIONI STANDARD 114
APPENDICE B SOMMARIO DEGLI OPERATORI 117
APPENDICE C TAVOLE 118
APPENDICE D SINTASSI 119
APPENDICE E NUMERAZIONE ERRORI 127
APPENDICE F ESEMPI DI PROGRAMMAZIONE 131

INDICE ANALITICO 141

RAPPORTO 141

1. Introduzione 141

2. Sommario del linguaggio 142

3. Notazioni, terminologia, vocabolario 145

4. Identificatori, numeri, stringhe 146

5. Definizione delle costanti 147

6. Definizione dei tipi dei dati 147
6.1 Tipi semplici 147
6.2 Tipi strutturati 148
6.3 Tipi puntatore 151

7. Dichiarazioni e denotazioni di variabili 152
7.1 Variabili complete 153
7.2 Variabili componenti 153

8. Espressioni 154
8.1 Operatori 155
8.2 Designatori funzionali 157

9. Istruzioni 158
9.1 Istruzioni semplici 158
9.2 Istruzioni strutturate 160

10. Dichiarazioni di procedure 165
10.1 Procedure standard 168

11. Dichiarazioni di funzioni 169
11.1 Funzioni standard 171

12. Input e output 172
12.1 Procedura read 173
12.2 Procedura readln 173
12.3 Procedura write 173
12.4 Procedura writeln 174
12.5 Procedure supplementari 175

13. Programmi 175

14. Standards implimentativi per la portabilità 176

15. Indice analitico 177

INTRODUZIONE

Molta parte del testo che segue è orientata verso un lettore che ha una minima conoscenza della terminologia degli elaboratori e che "sente" la struttura di un programma. Lo scopo di questa sezione è di ravvivare quell'intiuito.

{Programma 0.1
Assumendo un tasso d'inflazione annuale del 7,8 e 10%, trovare il fattore di svalutazione del franco, dollaro, marco, fiorino o lira in 1,2 ... n anni.}

```
program inflazione (output);
const n = 10;
var i : integer; w1, w2, w3 : real;
begin i := 0; w1 := 1.0; w2 := 1.0; w3 := 1.0;
  repeat i := i+1;
    w1 := w1 * 1.07;
    w2 := w2 * 1.08;
    w3 := w3 * 1.10;
    writlen (i, w1, w2, w3)
  until i=n
end.
```

1	1.070000000000e+00	1.080000000000e+00	1.100000000000e+00
2	1.144900000000e+00	1.166400000000e+00	1.210000000000e+00
3	1.225043000000e+00	1.259712000000e+00	1.331000000000e+00
4	1.310796010000e+00	1.360488960000e+00	1.464100000000e+00
5	1.402551730700e+00	1.469328076800e+00	1.610510000000e+00
6	1.500730351849e+00	1.586874322944e+00	1.771561000000e+00
7	1.1605781476478+00	1.713824268779e+00	1.948717100000e+00
8	1.718186179832e+00	1.850930210282e+00	1.143588810000e+00
9	1.8938459212420+00	1.999004627104e+00	1.357947691000e+00
10	1.967151357290e+00	1.158924997173e+00	1.593742460100e+00

Un *algoritmo* o programma è costituito da due parti: una descrizione delle *azioni* che devono essere fatte ed una descrizione dei *dati*, che sono modificati da queste azioni. Le azioni sono descritte da *istruzioni* ed i dati sono descritti da *dichiarazioni* e *definizioni*.

Il programma è diviso in un'intestazione ed un corpo, chiamato *blocco*. L'intestazione è costituita dal nome del programma e dai suoi parametri.

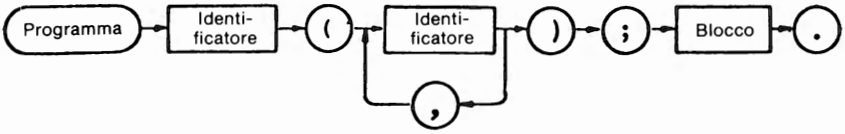
(Questi sono variabili (flussi) e rappresentano l'argomento ed i risultati dell'elaborazione. Si veda il capitolo 13.) Il flusso "output" è un parametro obbligatorio. Il blocco è costituito da sei sezioni che, eccetto l'ultima, possono essere vuote. Nell'ordine richiesto sono:

<parte dichiarazione etichette>
<parte definizione costanti>
<parte definizione tipo>
<parte dichiarazione variabili>
<parte dichiarazione funzioni e procedure>
<parte istruzioni>

La prima sezione elenca tutte le etichette definite in questo blocco. La seconda sezione definisce i sinonimi per le costanti: introduce cioé degli identificatori che possono, in seguito, essere usati in luogo di quelle costanti. La terza contiene la definizione dei tipi e la quarta quella delle variabili. La quinta sezione definisce le parti subordinate del programma (procedure e funzioni). La parte relativa alle istruzioni specifica le azioni che devono essere fatte.
Con una *carta sintattica* si può esprimere in modo più conciso lo schema di programma riportato sopra. Partendo dal grafo "programma" di figura 0.a, un percorso lungo il grafo stesso definisce un programma sintatticamente corretto.

Ciascun nodo fa riferimento ad un grafo con quel nome, che è poi usato per definire il suo significato. I simboli terminali (quelli effettivamente scritti in un programma Pascal) sono in nodi ovoidali. (Si veda l'appendice D per la carta sintattica completa di Pascal).

Programma



Blocco

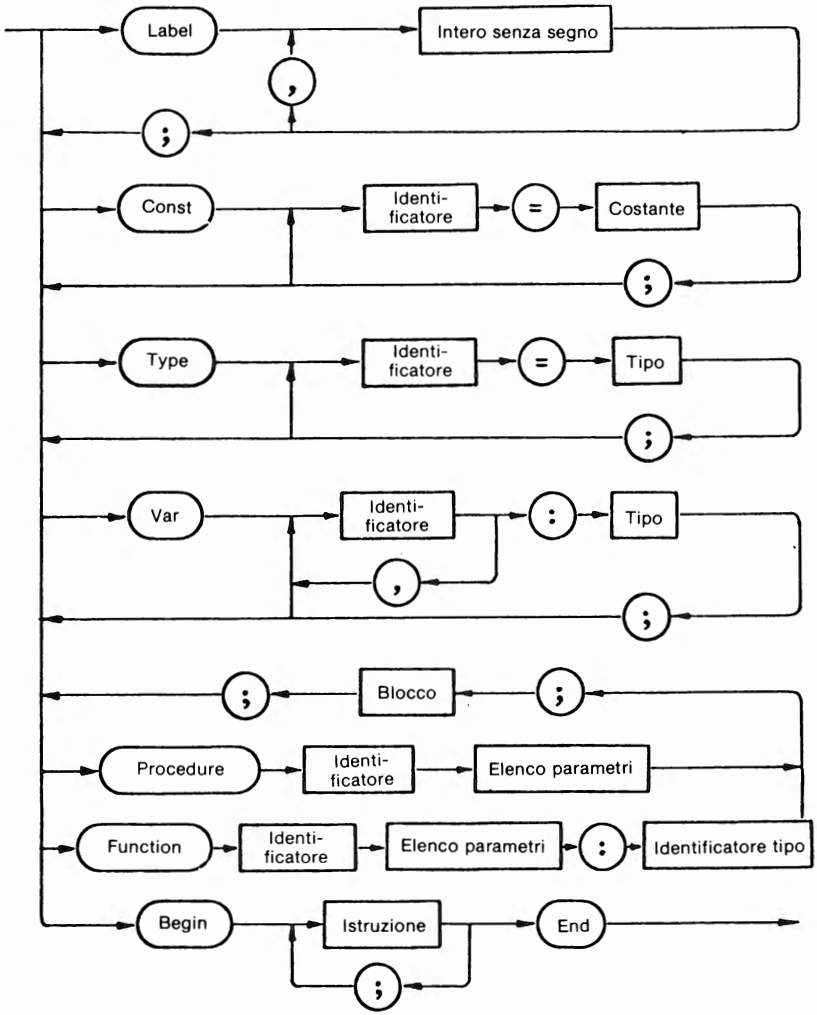
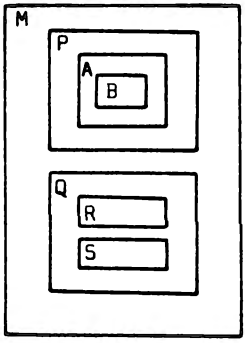


Fig. 0.a - Carta sintattica di definizione della struttura generale di un programma.

Una formulazione alternativa della sintassi è la forma tradizionale di **BACKUS-NAUR**, in cui i costrutti sintattici sono denotati da termini italiani racchiusi tra presenti angolari < e >. Questi termini richiamano alla mente la natura o il significato del costrutto. Il racchiudere una sequenza di costruttori nelle meta-parentesi { e } implica la ripetizione degli stessi zero o più volte (per la forma BNF di Pascal si veda l'appendice D). Si riporta come esemplificazione il costrutto < programma > di figura 0.a:

<programa> ::= <intestazione programma> <blocco>.
<intestazione programma> ::= **program** <identificatore>
(<identificatore flusso>
{ , <identificatore flusso> })
<identificatore flusso> ::= <identificatore>

Le formule sopra riportate sono chiamate "produzioni". Ogni procedura (funzione) ha una struttura simile ad un programma; ognuna è costituita da un'intestazione e da un blocco. Le procedure possono essere dichiarate dentro altre procedure (nidificazione). Etichette, sinonimi per costanti, tipi, variabili e dichiarazioni di procedure sono *locali* alla procedura in cui sono dichiarati. Questo significa che i loro identificatori hanno significato solo all'interno del testo del programma che costituisce la dichiarazione della procedura e che è chiamato *ambiente* ("scope" in inglese) di questi identificatori. Potendo nidificare le procedure, lo possono anche gli ambienti. Gli oggetti che sono dichiarati nel programma principale - non locali per alcuna procedura - sono chiamati *globali* e sono significativi nell'intero programma.



dove livello 0 = M
livello 1 = P,Q
livello 2 = A,R,S
livello 3 = D

Fig. 0.b - Struttura a blocchi.

Poiché le dichiarazioni di funzioni e procedure possono far nidificare un blocco dentro un altro, si riesce ad assegnare un livello di nidificazione ad ognuno. Se il livello esterno - blocco del programma principale - è indicato con livello 0, il blocco definito dentro questa sarà il livello 1; un blocco definito al livello i sarà (i+1). La figura 0.b mostra una struttura a blocchi.

Secondo questa formulazione, l'ambiente o campo di validità di un identifica tore x è l'intero blocco nel quale x è definito, includendo anche quei blocchi definiti nello stesso blocco in cui è definito x.

oggetti definiti nel blocco	sono accessibili nei blocchi
M	M,P,A,B,Q,R,S
P	P,A,B
A	A,B
B	B
Q	Q,R,S
R	R
S	S

In questo esempio tutti gli identificatori devono essere distinti. La sezione 3.e tratta il caso in cui gli identificatori non devono necessariamente essere distin ti.

Per i programmatori che conoscono ALGOL, PL/1, o FORTRAN, può essere utile dare un'occhiata a PASCAL con riferimento a quei linguaggi. A questo scopo elenchiamo le seguenti caratteristiche di Pascal:

1. La dichiarazione delle variabili è obbligatoria.
2. Certe parole chiave (p.e. **begin, end, repeat**) sono riservate e non possono essere usate come identificatori. In questo manuale sono in neretto.
3. Il punto e virgola (;) è considerato un separatore di istruzioni e non un terminatore (come per esem. in PL/1).
4. I tipi di dati standard sono: i numeri reali ed interi, i valori logici ed i caratteri (stampabili). Le strutture dati base includono: i vettori, i records (corrispondono alle "structure" in COBOL e PL/I), sets ed i flussi (se quenziali). Queste strutture possono essere combinate per formare vettori di sets, flussi di records, ... I dati possono essere allocati dinamicamente ed individuati con puntatori. Questi puntatori permettono una gestione completa delle liste. C'é la possibilità di dichiarare nuovi tipi base con costanti simboliche.
5. La struttura dati **set** offre possibilità simili alla "bit string" in PL/1.
6. I vettori possono essere di dimensioni arbitrarie con limiti qualsivoglia purché costanti: non ci sono cioé vettori dinamici.

7. Come in FORTRAN, ALGOL, PL/1, c'é l'istruzione di goto. Le etichette sono interi senza segno e devono essere dichiarate.
8. L'istruzione composta è quella dell'ALGOL e corrisponde al gruppo DO in PL/1.
9. La possibilità dello "switch" in ALGOL ed il goto calcolato del FOR TRAN sono implementati nell'istruzione **case**.
10. L'istruzione FOR, corrispondente al DO del FORTRAN, può solo avere incrementi di 1 (**to**) o -1 (**downto**) ed è eseguita solo fino a quando i valori della variabile di controllo sono dentro i limiti. L'istruzione controllata può non essere mai eseguita.
11. Non ci sono espressioni condizionali ed assegnamenti multipli.
12. Le procedure e le funzioni possono essere chiamate in modo recursivo.
13. Non ci sono come in ALGOL, attributi di "own" per le variabili.
14. I parametri sono chiamati per valore o per riferimento; non ci sono chiamate per nome.
15. La struttura del blocco differisce da quella dell'ALGOL e PL/1 per il fatto che non ci sono blocchi anonimi: ad ogni blocco è dato un nome ed è perciò costruito in una procedura.
16. Tutti gli oggetti - costanti, variabili, ... - devono essere richiamati prima di far loro riferimento. Sono tuttavia ammesse le 2 eccezioni seguenti:
    1) l'identificazione del tipo in un definizione del tipo puntatore (capitolo 10)
    2) chiamate a funzioni e procedure quando c'é un riferimento in avanti (capitolo II).

Dopo un primo approccio a PASCAL, parecchi tendono a lamentare l'assenza di certe comode caratteristiche; per esempio: un operatore esponenziale, con catenamento di stringhe, vettori dinamici, operazioni aritmetiche su valori booleani, conversione automatica dei tipi e dichiarazioni per difetto. Ma queste sono state omissioni deliberate e non delle sviste. In certi casi la loro presenza sarebbe stata, anzitutto, un invito a soluzioni inefficienti; in altri, si è ritenuto che sarebbero stati in contrasto con il fine di ottenere programmi chiari, affidabili e in "buon stile". Si è, infine, operata una cernita rigorosa nell'immensa varietà di costruttori disponibili in programmazione anche per rendere il compilatore relativamente compatto ed efficiente: efficiente ed economico sia per gli utenti che scrivono solo programmi brevi usando pochi tipi di costruttori sia per coloro i quali scrivono dei grossi programmi e tendono ad usare tutte le possibilità offerte dal linguaggio.

CAPITOLO 1

NOTAZIONE E VOCABOLARIO

Il *vocabolario* base è costituito dai simboli base classificati in lettere, cifre e simboli speciali. I *simboli spaciali* sono gli operatori e i delimitatori:

+	:	(	**and**	**end**	**nil**	**set**
-	'	)	**array**	**file**	**not**	**then**
*	=	[	**begin**	**for**	**of**	**to**
/	<>	]	**case**	**function**	**or**	**type**
:=	<	}	**const**	**goto**	**packed**	**until**
.	<=	{	**div**	**if**	**procedure**	**var**
,	>=	↑	**do**	**in**	**program**	**while**
;	>	..	**downto**	**label**	**record**	**with**
			else	**mod**	**repeat**	

Le parole delimitatrici (o parole riservate) vengono normalmente sottolineate nei programmi scritti a mano per enfatizzare la loro interpretazione come simboli singoli con significato fisso. Il programmatore non può usare queste parole se non in un contesto come quello specificato nella definizione di Pascal; in particolare queste parole non possono essere usate come identificatori. Sono scritte coma una sequenza di lettore senza caratteri di fuga (escape characters).

Il costrutto:

{ < qualunque sequenza di simboli non contenenti"}">}

può essere inserito tra due qualsiasi identificatori, numeri, o simboli. Viene chiamato *commento* e può essere tolto dal testo del programma senza che il significato cambi. I simboli { e } non ricorrono con altro significato nel linguaggio e quando compaiono in descrizioni sintattiche, denotano dei meta-simboli come | e ::=. (Nei sistemi in cui non sono disponibi li le parentesi graffe, è usata al loro posto la coppia di caratteri (* e *).

Gli *identificatori* sono nomi che denotano costanti, tipi, variabili, procedure e funzioni. Devono iniziare con una lettura che può essere seguita da qualsiasi

combinazione e numero di lettere e cifre. Anche se un identificatore può essere molto lungo, le implementazioni di Pascal standard riconosceranno sempre, come significativi, i primi 8 caratteri di un identificatore. Identificatori denota nti oggetti diversi devono quindi differire nei loro primi otto caratteri.

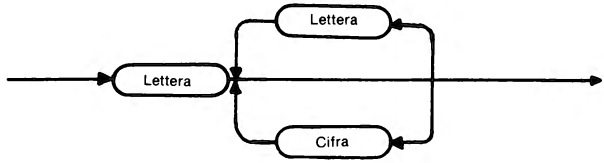


Fig. 1.a - Identificatore

esempi di identificatori corretti:

somma radice3 pi h4g x
questopuoessereunidentificatorelegaleanchesemoltolungo
questopuoessereunidentificatoremaprobabilmenteelostessodiprima

identificatori non corretti:

3rd array livello.4 radice-3

Certi identificatori, chiamati *identificatori standard*, sono predefiniti (p.e. sin, cos). Diversamente dalle parole deliminatrici (p.e. array), non si è vincolati a questa definizione e si può scegliere di ridefinire ogni identificatore standard: è come se questi fossero dichiarati in un ipotetico blocco che circonda l'intero blocco del programma.

Per i *numeri* è usata la notazione decimale. La lettera E quando precede il fattore di scala è pronunciata come "n volte 10 elevato a". La sintassi di un numero senza segno è riassunta in figura 1.b, mentre in figura 1.c l'intero senza segno.

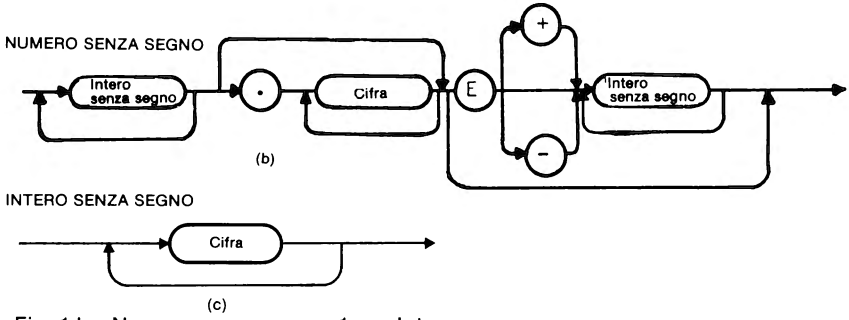


Fig. 1.b - Numero senza segno, 1. c - Intero senza segno.

Si noti che se il numero contiene il punto decimale, almeno una cifra deve precedere e seguire il punto; inoltre, nei numeri non ci può essere la virgola.

numeri senza segno:

3 03 6272844 0.6 5E-8 49.22E+08 1E10

numeri scritti non correttamente:

3,487,159 XII .6 E10 5.E-16

Spazi, fine linea e commenti sono considerati *separatori*. Un numero qualsivo glia di separatori può stare tra due qualunque simboli Pascal consecutivi, con le seguenti eccezioni: all'interno di identificatori, numeri o simboli speciali non possono esserci separatori. Deve, tuttavia, esserci almeno un separatore tra ogni coppia di identificatori, numeri o parole simboli consecutivi.

Una sequenza di caratteri racchiusa tra apici è chiamato *stringa*. Per includere un apice in una stringa lo si deve scrivere due volte.

esempi di stringhe:

'a' ';' '3' 'begin' 'don' 't'
'questa stringa ha 31 caratteri'

CAPITOLO 2

IL CONCETTO DI DATO

Dato è la parola che indica in generale tutto ciò su cui agisce l'elaboratore. A livello hardware o di codice macchina, tutti i dati sono rappresentati come sequenza di cifre binarie (bits). Linguaggi di più alto livello ci permettono di usare astrazioni e di ignorare i dettagli della rappresentazione: soprattutto grazie allo sviluppo del concetto di *tipo di dato*. Il tipo di dato definisce l'insieme dei valori che una variabile può assumere. Ogni variabile di un programma deve essere associata con uno ed un solo tipo. Sebbene i tipi di dato in Pascal possono essere molto sofisticati, ciascuno deve essere costruito partendo da tipi non strutturati. Un tipo non strutturato è o definito dal programmatore, e quindi chiamato di tipo scalare dichiarato, oppure è uno dei quattro tipi scalari standard: integer, real, boolean, char.

Un tipo scalare è caratterizzato dall'insieme dei suoi valori distinti, sui quali è definito un ordinamento lineare. I valori sono denotati dagli identificatori inseriti nella definizione del tipo (si veda il capitolo 5).

A. Il tipo booleano (boolean)

Un valore booleano è uno dei valori della tabella della verità denotato dagli identificatori predefiniti false e true.

I seguenti opeatori logici danno un valore booleano quando sono applicati ad operandi di tipo booleano: (Nell'appendice b sono riassunti tutti gli operatori)

And	unione logica
or	disgiunzione logica
not	negazione logica

Tutti gli operatori di relazione (=, <>, <=, <, >, >=, **in**) danno un valore booleano. Il tipo booleano è definito in modo tale che false è < di true. È perciò possibie definire ognuna delle 16 operazioni booleane usando gli operatori logici e di relazione riportati sopra. Per esempio, se p e q sono valori booleani, si può scrivere

l'implicazione	come	p <= q
l'or esclusivo	come	p <> q
l'equivalenza	come	p = q

Le funzioni booleane standard - vale a dire le funzioni standard che danno un risultato booleano - sono:

odd(x)	true se l'intero x è dispari, altrimenti false
eoln(f)	fine linea (spiegato nel capitolo 9)
eof(f)	fine flusso (spiegato nel capitolo 9)

B. Tipo intero (Integer)

Un valore tipo intero è un elemento del sottoinsieme dei numeri interi definito all'implementazione.

I seguenti operatori aritmetici danno un valore intero quando sono applicati ad operandi interi:

*	moltiplicazione
div	divisione con troncamento: il valore non è arrotondato
mod	a **mod** b = a- ((a **div** b)*b)
+	addizione
–	sottrazione

Gli operatori di relazione =, <>, <, <=, >=, > danno un risultato booleano quando sono applicati ad operandi interi; <> sta ad indicare disuguaglianza.

Quattro importanti funzioni standard che danno risultati interi sono:

abs(x)	il risultato è il valore assoluto di x
sqr(x)	il risultato è x al quadrato
trunc(x)	x è un valore reale; il risultato è la sua parte intera. (la parte frazionale è scartata; p.e. trunc (3.7) = 3 e trunc (-3.7) = -3)
round(x)	x è un valore reale; il risultato è l'intero arrotondato, round(x) significa per x > = 0 trunc (x+0.5), e per x<0 trunc (x-0.5)

Note: abs e sqr danno un risultato intero solo quando anche i loro argomenti sono di tipo intero. Se i è una variabile di tipo intero allora:

succ(i)	dà il "prossimo" intero, e
pred(i)	dà l'intero precedente.

Questo, tuttavia, può essere reso più chiaramente dalle espressioni:
i+1 e i-1

Esiste un identificatore standard *maxint* dipendente dall'implementazione. Se a e b sono espressioni intere si garantisce che l'operazione

a **op** b

è correttamente implementata quando:

abs (a **op** b) <= maxint,
abs(a) <= maxint e
abs(b) <= maxint.

C. Il tipo reale (real)

Un valore di tipo reale è un elemento del sottoinsieme dei numeri reali definito all'implementazione.

Se almeno uno degli operandi è di tipo reale (l'altro può essere intero) i seguenti operatori danno un valore reale:

* moltiplicazione
/ divisione (entrambi gli operandi possono essere interi, ma il risultato è sempre reale)

\+ addizione
− sottrazione

Funazioni standard che con argomento reale danno un risultato reale sono:

abs(x) valore assoluto
sqr(x) x al quadrato

Funzioni standard che con argomento intero o reale danno risultato reale sono:

sin(x) funzioni trigonometriche
cos(x)
arctan(x)
ln(x) logaritmi naturali
exp(x) funzione esponenziale
sqrt(x) radice quadrata

Attenzione: sebbene i reali sono inclusi nei tipi scalari, non possono essere usati nello stesso contesto degli altri tipi scalari. In particolare, le funzioni pred e succ non possono avere argomenti reali; valori di tipo reale non possono essere usati come indici di vettori, né nel controllo di istruzioni di tipo for, né per definire il tipo base di un set.

D. Il Tipo carattere (Char)

Un valore di tipo char è un elemento di un insieme finito e ordinato di caratteri. Ogni sistema definisce un tale insieme per le comunicazioni. Questi caratteri sono poi disponibili sui dispositivi di ingresso ed uscita. Sfortunata mente non esiste un insieme di carattere standard; perciò la definizione degli elementi ed il loro ordinamento è strettamente vincolato all'implementazione.

Le seguenti assunzioni minime, indipendentemente dall'implementazione fatta, sono valide per il tipo char:

L'insieme dei caratteri include

1. l'insieme delle lettere maiuscole latine in ordine alfabetico A ...Z
2. l'insieme contiguo e numericamente ordinato delle cifre decimali 0 ... 9
3. il carattere "spazio"

Un carattere racchiuso tra apici denota una costante di questo tipo.

Esempi:

'*' 'G' '3' '''' 'X'

(per rappresentare un apice lo si scrive due volte.)

Le due funzioni standard *ord* e *chr* permettono di mappare l'insieme dei caratteri su un sottoinsieme dei numeri naturali (chiamati i numeri ordinali dell'insieme dei caratteri) e viceversa; ord e chr sono chiamate *funzioni* di *trasferimento*.

ord(c) è il numero ordinale del carattere c nell'insieme ordinato dei caratteri. (si veda anche la sezione 5.A)

chr(i) è il carattere con il numero ordinale i

Si vede immediatamente che ord e chr sono funzioni diverse:

chr(ord(c))=c e ord(chr(i))=i

Inoltre l'ordinamento di un certo insieme di caratteri è definito da

c1 < c2 iff ord(c1) < ord(c2)

La definizione può essere estesa ad ognuno degli operatori di relazione: =, <>, <, <=, >=, >. Se R denota uno di questi operatori:

c1 R c2 iff ord(c1) R ord(c2)

Quando l'argomento delle funzioni standard pred e succ è di tipo char, le funzioni possono essere come:

pred(c) = chr(ord(c)-1)
succ(c) = chr(ord(c)+1)

Nota:Il predecessore (successore) di un carattere dipende dall'insieme dei caratteri implementari ed è indefinito se non esiste.

CAPITOLO 3

INTESTAZIONE DEL PROGRAMMA E SEZIONE DI DICHIARAZIONE

Ogni programma è costituito da un'intestazione e da un blocco. Il blocco contiene la parte dichiarativa, nella quale sono definiti tutti gli oggetti locali al programma, e la sezione istruzioni, che specifica le azioni che si eseguono su questi oggetti.

```
<programma< ::= <intestazione programma> <blocco>
<blocco> ::=    <parte dichiarazione etichette>
                <parte definizione costanti>
                <parte definizione tipi>
                <parte dichiarazione variabili>
                <parte dichiarazione procedure e funzioni>
                <parte istruzioni>
```

A. Intestazione programma

L'intestazione dà il nome al programma (nome che comunque non ha altri significati) ed elenca i parametri, con i quali il programma comunica con l'esterno (si veda il capitolo 13.B.1).

```
<intestazione programma> ::= programm <identificatore>
                             ( <identificatore flusso>
                             {, <identificatore flusso> } );
```

B. Parte dichiarazione etichette

Ogni istruzione in un programma può avere un'etichetta seguita da due punti che rende possibile i riferimenti per istruzioni di goto. L'etichetta va definita nella *parte dichiarazione etichette* prima del suo uso. Il simbolo **label** individua questa parte ed ha la forma:

```
label <etichetta> {, <etichetta> };
```

un'etichetta può essere un qualunque intero senza segno con al massimo quattro cifre.

esempio:

label 3,18;

C. Parte definizione costanti

Una *definizione costanti* introduce un identificatore come sinonimo per una costante. Il simbolo const individua la parte di definizione delle costanti che ha la seguente forma:

const <identificatore> = <costante>; { <identificatore> = <costante>;}

dove una costante è o un numero o un identificatore costante (anche con segno) o una stringa.

L'uso di identificatori per le costanti rende uun programma più leggibile ed aiuta nella documentazione. Permette inoltre al programmatore di raggrup pare all'inizio del programma, dove possono facilmente essere notati e/o cambiate, le quantità dipendenti dalla macchina o dall'esempio. (Questo incrementa la polarità e la modularietà del programma).

Come esempio si consideri il seguente programma:

```
{ programma 3.1
esempio di parte di definizione costanti }

program conversione (output);


const aggiungi = 32; molper = 1.8; basso = 0;  alto = 39; separatore = '-------';
var grado: basso..alto;
begin
     writeln (separatore);
     for grado: = basso to alto do
     begin write (grado, 'c', round (grado*molper+aggiungi), 'f');
       if odd (grado) then writeln
     end
     writeln;
     writeln (separatore)
end.
```

0c	32f	1c	34f
2c	36f	3c	37f
4c	39f	5c	41f
6c	43f	7c	45f
8c	46f	9c	48f
10c	50f	11c	52f
12c	54f	13c	55f
14c	57f	15c	59f
16c	61f	17c	63f
18c	64f	19c	66f
20c	68f	21c	70f
22c	72f	23c	73f
24c	75f	25c	77f
26c	79f	27c	81f
28c	82f	29c	84f
30c	86f	31c	88f
32c	90f	33c	91f
34c	93f	35c	95f
36c	97f	37c	99f
38c	100f	39c	102f

D. Parte definizione tipo

In Pascal il tipo del dato può essere descritto direttamente nella dichiarazione della variabile o caratterizzato da un *identificatore di tipo*. Vengono forniti non solo parecchi identificatori di tipo standard, ma anche un meccanismo, la *definizione del tipo*, per crearne di nuovi. Il simbolo **type** individua la parte di programma contenente la definizione del tipo. La definizione stessa determina un insieme di valori ed associa un identificatore con l'insieme. La forma generale è:

type <identificatore> = <tipo>; { < identificatore> = <tipo>; }

Esempi di definizione di tipo possono essere trovati nei capitoli seguenti.

E. Parte dichiarazione variabili

Ogni variabile che compare in un'istruzione deve essere dichiarata in una *dichiarazione di variabile* e la dichiarazione deve precedere qualsiasi uso della stessa.

Una dichiarazione associa un identificatore ed un tipo di dato con una nuova

variabile: basta elencare l'identificatore seguito dal tipo. Il simbolo **var** va posto in testa alla parte di dichiarazione delle variabili. La forma generale è:

```
var   <identificatore> {, <identificatore>} : <tipo>;
      { <identificatore> {, <identificatore>} : <tipo>;}
```

esempio:

```
var   radice1, radice2, radice3: real;
      contat,i : integer;
      trovato : boolean;
      riempi : char;
```

L'associazione tipo/identificatore è valida in tutto il blocco contenente la dichiarazione, a meno che l'identificatore sia ridefinito in un blocco subordinato. Supponiamo che il blocco B sia interno al blocco A: sia cioé dichiarato dentro l'ambiente di A e perciò subordinato ad A, come in fig. O.b. È possibile dichiarare in B un identificatore che è già stato dichiarato in A. In questo modo si associa l'identificatore con una variabile locale a B - non disponibile per A - che può essere di qualsiasi tipo. La seconda definizione è quindi valida in tutto l'ambiente di B, a meno che ci sia una nuova dichiarazione in un blocco subordinato a B. Non è concessa la dichiarazione di un unico identificatore più di una volta dentro lo stesso livello ed ambiente. Perciò le dichiarazioni che seguono sono sempre scorrette:

```
var   a :integer;
      a :real;
```

F. Parte dichirazione procedure e funzioni

Ogni procedura o funzione deve essere definita - o dichiarata prima del suo uso. La dichiarazione di procedure e funzioni è trattata nel capitolo II. Le procedure sono sottoprogrammi chiusi e sono attivate da istruzioni di proce dura. Le funzioni sono sottoprogrammi chiusi che danno come risultato un valore; possono perciò essere usate come elementi costituenti espressioni.

CAPITOLO 4

IL CONCETTO DI AZIONE

Essenziale per un programma è l'azione: un programma deve fare qualcosa con i suoi dati. Questa azione potrebbe essere anche la scelta di non fare niente! Le azioni sono descritte da *istruzioni*. Le istruzioni possono essere semplici (p.e. la istruzione di assegnamento) o *strutturate*.

A. Istruzione di assegnamento

L'istruzione di *assegnamento* è da considerarsi l'istruzione fondamentale. Specifica che un nuovo valore calcolato va assegnato ad una variabile. La forma di un assegnamento è:

<variabile> := <espressione>

dove := *è l'operatore assegnamento* che non va confuso con l'operatore di relazione =. L'istruzione "a := 5" va letta come: "il valore corrente di a è sostituito con il valore 5", o semplicemente, "a *diviene* 5".

Il nuovo valore è ottenuto computando un'*espressione* costituita da operandi costanti o variabili, operatori e designatori di funzioni. (Un disignatore di funzione specifica l'attivazione della funzione. Le funzioni standard sono elencate nell'appendice A; la definizione di funzioni da parte dell'utente è spiegata nel capitolo 11.) Un'espressione è una regola per calcolare dei valori per la quale sono osservate le regole convenzionali di valutazione: valutazione da sinistra a destra e regole di *precedenza* degli *operatori*. L'operatore **not** (applicato ad un operando booleano) ha la precedenza più alta ed è seguito dagli operatori moltiplicativi (*, /, **div**, **mod**, **and**), dagli operatori additivi (+, -, **or**) ed infine a più bassa priorità dagli operatori di relazione (=, <>, <, >, <=, >=, **in**). Ogni espressione tra parentesi è valutata in modo indipendente dagli operatori che la precedono o la seguono.

esempi:

2 * 3-4 * 5	= (2 * 3) - (4 * 5)	= -14
15 **div** 4 * 4	= (15 **div** 4) * 4	= 12
80/5/3	= (80/5) /3	= 5.333
4/2 * 3	= (4/2) * 3	= 6.000
sqrt (sqr(3)+11*5)	=	= 8

La specifica delle regole di precedenza è riportata nell'appendice D: il lettore deve farvi riferimento per ogni dubbio.
Le espressioni booleane hanno la proprietà che i loro valori possono essere conosciuti prima che sia stata calcolata tutta l'espressione. Si supponga di avere una espressione del genere:

(x > 0) **and** (x < 10)

se x è = 0 il valore dell'espressione è già conosciuto dopo il calcolo del primo termine e perciò non è necessario calcolare il secondo. In casi come questo, le regole di Pascal, né esigono né vietano la valutazione della seconda parte. Il programmatore deve quindi assicurarsi che il secondo fattore sia definito correttamente e questo indipendentemente dal valore del primo. Perciò, se si assume che un vettore ha un indice che va da 1 a 10, l'esempio che segue è errato:

```
x := 0;
repeat x := x+1 until (x > 10) or (a [x]=0)
```

(È evidente che se nessun a [i] = 0, il programma farà riferimento ad un elemento a [11].)

I flussi sono le sole variabili alle quali non è possibile fare assegnamenti. La variabile (o la funzione) e l'espressione devono essere dello stesso tipo eccet tuato quando la variabile è di tipo reale: in questo caso l'espressione può essere intera. (Se si ha a che fare con una variabile di tipo sottocampo, la validità dell'assegnamento è determinata dallo scalare associato; si veda la sezione 5.d).

esempi di assegnamento:

```
radice1   := pi*x/y
radice1   := -radice1
radice3   := (radice1 + radice2) * (1.0 + y)
trovato   := y > w
contat    := contat + 1
```

```
grado    := grado + 10
sqrpr    := sqr (pr)
y        := sin (x) + cos (y)
```

B. Istruzione composta

L'istruzione composta specifica che le istruzioni che la compongono vanno eseguite nella sequenza in cui sono scritte. I simboli **begin** ed **end** fanno da parentesi per l'istruzione. Si noti che il "corpo" di un programma ha la forma di un'istruzione composta.

```
{ programma 4.1
  l'istruzione composta}

program beginend (output);

var somma : integer;
begin
    somma := 3 + 5;
    writlen (somma, -somma)
end.
```

```
8        -8
```

Pascal usa il punto e virgola per *separare* le istruzioni, non per terminarle: il punto e virgola non fa quindi parte della istruzione. Per quanto riguarda il punto e virgola si vedono le regole esplicitate nella sintassi dell'appendice D. Se si fosse scritto un punto e virgola dopo la seconda istruzione sarebbe stato come introdurre una *istruzione vuota* (che implica nessuna azione) tra il punto e virgola ed il simbolo **end**. Il fatto che in questo punto sia permessa un'istru zione vuota non nuoce, ma, come è mostrato nell'esempio della sezione 4.D, dei punti e virgola posizionati male possono causare dèi guai.

C. Istruzioni di iterazione

Le *istruzioni iterative* specificano che certe istruzioni vanno eseguite ripetuta mente. Se il numero di ripetizioni è conosciuto in anticipo (prima che le ripetizioni siano iniziate), l'istruzione che in modo più appropriato esprime la situazione è l'istruzione di for; nell'altro caso si usa l'istruzione repeat o while.

C.1 L'istruzione while

L'istruzione while ha la forma:

while <espressione> **do** <istruzione>

L'espressione che controlla la ripetizione deve essere di tipo booleano ed è valutata prima di ogni iterazione cosicché conviene aver cura di rendere l'espressione il più semplice possibile.

```
{ programma 4.2
  calcola h (n) = 1 + 1/2 + 1/3 + ... + 1/n}

program perwhile (input, output);

var n : integer; h : real;
begin read (n); write (n);
    h := 0;
    while n > 0 do
       begin h := h + 1/n; n := n-1
       end;
    writeln (h)
end.
          10      2.928968253968e+00
```

L'istruzione associata al while (un'istruzione composta nell'esempio ripor tato) è ripetuta fino a che l'espressione diviene falsa. Nel caso in cui, già all'inizio, il valore sia falso, l'istruzione non viene eseguita.

C. 2 L'istruzione repeat

L'istruzione repeat ha la forma:

repeat <istruzione> {; <istruzione>} **until** <espressione>

La sequenza di istruzioni tra i simboli **repeat** e **until** è eseguita almeno una volta. Le esecuzioni ripetute sono controllate dall'espressione booleana, che é valutata dopo ogni iterazione.

```
{ programma 4.3
  calcola h (n) = 1 + 1/2 + 1/3 + ... + 1/n }
program perrepeat (input, output);
var n : integer; h : real;
begin read (n); write (n);
     h := 0;
     repeat h := h + 1/n; n := n-1
     until n = 0;
     writeln (h)
end.

          10      2.928968253968e+00
```

Il programma è corretto per n > 0 ma si consideri cosa succede per n<=0. La versione del programma con while è corretta per ogni n, incluso n=0.

Si noti che l'istruzione di repeat esegue una sequenza di istruzioni per cui le parentesi **begin ... end** sarebbero ridondanti anche se non scorrette.

C.3 L'istruzione for

L'istruzione for indica che un'istruzione deve essere ripetutamente eseguita mentre una progressione di valori è assegnata alla *variabile di controllo* dell'i struzione for. La sua forma generale è:

```
    for   <variabile di controllo> :=  <valore iniziale> to
                                       <valore finale> do <istruzione>
(oppure)
    for   <variabile di controllo   > := <valore iniziale> downto
                                       <valore finale> do <istruzione>
```

```
{ programma 4.4
  calcola h(n) = 1 + 1/2 + 1/3 + ... + 1/n }

program    perfor (input, output);

var    i,n integer; h: real;
begin    read (n); write (n);
     h := 0;
     for    i := n downt 1 do h := h + 1/i;
     writeln (h)
end.

          10      2.928968253968e+00

{ programma 4.5
  calcolo il coseno usando l'espansione:
     cos (x) = 1 - x ** 2/(2 *) + x ** 4/(4*3*2*1) - ... }
```

```
program    coseno (input, output);
const    eps = 1e-14;
var    x,sx,s,t : real;
       i,k,n, : integer;
begin    read (n);
    for    i := 1 to n do
    begin    read (x); t := 1; k := 0; s := 1; sx := sqr (x);
        while    abs (t) < eps*abs (s) do
        begin    k := k + 2; t := -t*sx/(k*(k-1));
            s := s+t
        end;
        writeln (x,s,k div 2)
    end
end.
```

```
1.534622222233e -01     9.882477647614e -01     5
3.333333333333e -01     9.449569463147e -01     6
5.000000000000e -01     8.775825618904e -01     7
1.000000000000e+00      5.403023058681e -01     9
3.141592653590e+00     -1.000000000000e+00     14
```

La variabile di controllo, il valore iniziale, il valore finale devono essere scalari - esclusi reali - dello stesso tipo e non devono essere modificati dall'istruzione for. Il valore iniziale e finale sono calcolati una sola volta. Se il valore iniziale è maggiore (**to**) o minore (**downto**) del valore finale, l'istruzione non viene eseguita. Il valore finale della variabile di controllo è indefinito dopo l'uscita normale da un for.

Un'istruzione for come la seguente:

```
for    v := e1 to e2 do S
```

è equivalente alla sequenza di istruzioni:

```
if    e1<=e2 then
begin    v := e1; S; v := succ (v); S; ...; v := e2; S
end
{ a questo punto, v è indefinito }
```

Come esempio si consideri il seguente programma.

```
{ programma 4.6
  calcolo 1 - 1/2 + 1/3-...+1/9999 - 1/10000, in 4 modi.
```

1) da sinistra a destra in successione
2) da sinistra a destra, tutti i positivi poi i negativi, quindi sottrai
3) da destra a sinistra in successione
4) da destra a sinistra, tutti i negativi poi i positivi, quindi sottrai}

```
program    somma (output);

var  s1, s2p, s2n, S3, s4p, s4n, lrp, lrn, rlp, rln : real; i : integer;
begin  s1 := 0; s2p := 0; s2n := 0; s3 := 0; s4p := 0; s4n := 0;
    for    i := 1 to 5000 do
    begin
        lrp  := 1/(2*i-1); { term pos, da sin a des }
        lrn  := 1/(2*i); { term neg, da sin a des }
        rlp  := 1/(10001-2*i): { term pos, da des a sin }
        rln  := 1/(10002-2*i); { term neg, da des a pos }
        s1   := s1 + lrp - lrn;
        s2p  := s2p + lrp; s2n := s2n + lrn;
        s3   := s3 + rlp - rln;
        s4p  := s4p + rlp; s4n := s4n + rln
    end;
    writeln (s1, s2p, - s2n);
    writeln (s3, s4p, - s4n)
end
```

```
6.930971830595e-0.1        6.930971830612e-01
6.930971830599e-0.1        6.930971830601e-01
```

Perché le quattro somme , "identiche", sono diverse?

D. Istruzioni condizionali

Un'istruzione condizionale - if o case - sceglie un'*unica* istruzione tra quelle da potersi eseguire. L'*istruzione if* specifica che una istruzione va eseguita solo se una certa condizione (espressione booleana) è vera. Se è falsa o non viene eseguita nessuna istruzione oppure quella che segue il simbolo **else**.

D.1 L'istruzione if

Le due forme dell'istruzione if sono :

if <espressione> **then** <istruzione>

(o)

if <espressione> **then** <istruzione> **else** <istruzione>

L'espressione tra i simboli **if** e **then** deve essere del tipo booleano. Si noti che la prima forma può essere considerata un'abbreviazione della seconda quando l'istruzione alternativa è vuota.
Attenzione: non c'é mai un punto e virgola prima di un **else**! Perciò, scrivere:

```
if p then begin S1; S2; S3 end; else S4
```

non è corretto. Ma ancor più errato è il testo:

```
if p then; begin S1; S2; S3 end
```

L'istruzione controllata dall'*if* è l'istruzione vuota tra **then** e il punto e virgola! L'istruzione composta che segue l'**if** sarà quindi sempre eseguita.

L'ambiguità sintattica che sorge dal costrutto:

```
if <espressione-1> then if <espressione-2> then <istruzione-1>
    else <istruzione-2>
```

è risolta con il costrutto equivalente:

```
if <espressione-1> then
    begin if <espressione-2> then <istruzione-1>
        else <istruzione-2>
    end
```

Il lettore tenga presente che delle istruzioni di if formulate con leggerezza possono essere molto dispendiose in termini di tempo di esecuzione. Si supponga, per esempio, di avere n condizioni matuamente esclusive, cl ... cn, ognuna delle quali inneschi un'azione diversa. Sia P(ci) la probabilità che ci sia vero, e si supponga che P(ci) >=P (cj) per i<j.
Allora la più efficiente sequenza di clausole è:

```
if cl then s 1
    else if c 2 then s 2
        else ...

                else if c (n-1) then s (n-1) else sn
```

L'adempimento di una condizione e l'esecuzione della sua istruzione completa l'if, cortocircuitando perciò la rimanente parte del testo.

Se "trovato" è una variabile di tipo booleano, un'altro frequente abuso dell'istruzione if può essere illustrato dal seguente esempio:

```
if a=b then trovato := true else trovato := false
```

Molto più semplicemente si può scrivere:

```
trovato := a=b
```

```
{ programma 4.7
  scrive numeri romani }

program    romani (output);

var    x,y  :integer;
begin    y := 1;
     repeat    x := y; write (x, ' ');
          while    x >= 1000 do
               begin    write ('m'); x := x-1000 end;
          if    x >= 500 then
               begin    write ('d') ; x := x-500   end;
          while    x >= 100   do
               begin    write ('c') ; x := x-100   end;
          if    x >= 50   then
               begin    write ('l') ; x := x-50    end;
          while    x >= 10    do
               begin    write ('x') ; x := x-10    end;
          if    x >= 5    then
               begin    write ('v') ; x := x-5     end;
          while    x  >= 1 do
               begin    write ('i') ; x := x-1     end;
          writeln;    y := 2*y
     until    y > 5000
end.
```

```
   1 i
   2 ii
   4 iiii
   8 viii
  16 xvi
  32 xxxii
  64 lxiiii
 128 cxxviii
 256 cclvi
 512 dxii
1024 mxxiiii
2048 mmxxxxviii
4096 mmmmlxxxxvi
```

Si noti che la clausola if controlla solo un'istruzione; bisogna quindi usare l'istruzione composta quando si vogliono far eseguire più azioni.

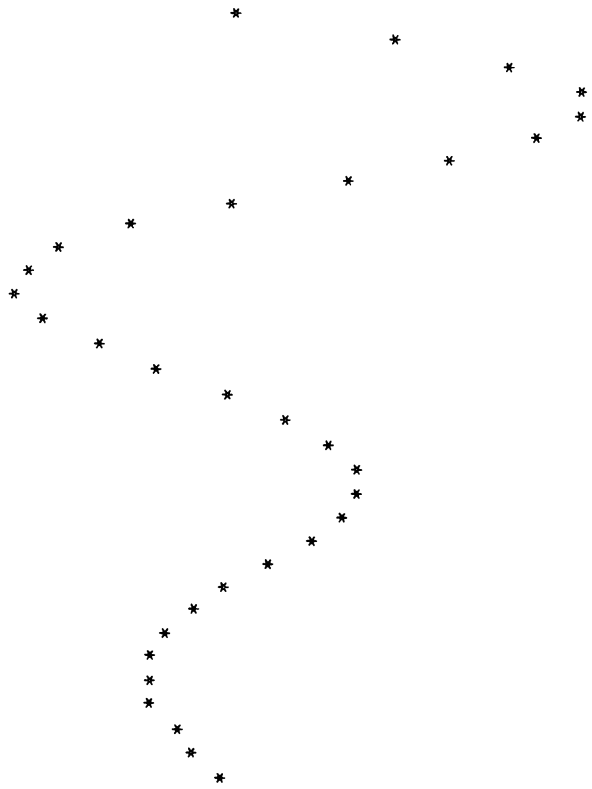
Il prossimo programma calcola la potenza di una base reale x elevata all'esponente intero - non negativo - y. Si può ottenere una versione più semplice e corretta mettendo l'istruzione while più interna: il risultato z è ottenuto con y moltiplicazioni di x. Si noti l'invariante di ciclo: z*(u**e)=**y. L'istruzione di while interna lascia z e u**e invariati e con questo migliora l'efficienza dell'algoritmo.

```
{ programma 4.8
  elevazione a pot. con esponente naturale }

program    elevpot (input, output);

var    e,y : integer; u,x,z : real;
begin    read (x,y); write (x,y);
     z := 1; u := x; e := y;
     while    e>0 do
     begin    { z*u**e = z**y, e>0 }
          while not    odd (e) do
               begin    e := e div 2; u := sqr (u)
               end;
          e := e-1; z := u*z
     end;
     writeln (z){ z = x**y }
end.
```

```
2.000000000000e+00                    7    1.280000000000e+02
```

Il programma che segue computa e rappresenta graficamente una funzione f(x). La variabile indipendente x, reale, corre lungo un'asse verticale ed i valori delle funzioni sono rappresentati con asterischi. Gli asterischi sono separati da tanti spazi quanti se ne ottengono moltiplicando f (x) per un fattore di scala s, arrotondando quindi il prodotto al prossimo intero ed aggiungendo infine una costante h.

```
{ programma 4.9
  rappresentazione grafica di una funzione
  f (x) = exp (-x) * sin (2*pi*x) }

program     graficol (output);
const   d = 0.0625; { 1/16, 16 linee per intervallo [ x,y+1 ] }
        s = 32; { 32 car. di larghezza per intervallo [ y,y+1 ] }
        h = 34; { posizione asse x }
        c = 6.28318; { 2* pi } lim = 32;
var   x,y real; i,n : integer;
begin
    for   i := 0 to lim do
        begin   x := d*i; y := exp (-x)*sin (c*x);
        n := round (s*y)+h;
        repeat   write (' '); n := n-1
        until   n=0;
        writeln ('*')
    end
end.
```



L'istruzione case (selezione) è costituita da un'espressione (il selettore) e da una lista di istruzioni, ognuna delle quali è etichettata con una costante dello stesso tipo del selettore. Il selettore deve essere di tipo scalare, reale escluso. L'istruzione di selezione manda in esecuzione l'istruzione la cui etichetta è uguale al valore corrente del selettore; se non esiste un'etichetta con quel valore, l'effetto non è definito. Dopo il completamento dell'istruzione scelta, il controllo è passato alla fine della selezione. La sua forma è:

```
case    <espressione> of
        <prima etichetta case> : <istruzione>;
                    ...
        <ultima etichetta case> : <istruzione>
end
```

esempi: (si supponga **var** i: integer; ch: char;)

```
case  i  of                      case  ch  of
    0: x := 0;                       'a', 'b', 'c' : ch := succ(ch);
    1: x := sin(x);                  'd', 'e' :      ch := pred(ch);
    2: x := cos(x);                  'f', 'g' :      < selezione nulla >
    3: x := exp(x);              end
    4: x := ln(x)
end
```

Si noti che le "etichette del case" *non* sono etichette ordinarie a cui si possa far riferimento con un goto (si veda la sezione 4.E). L'ordinamento può essere qualunque ma non ne possono esistere due eguali all'interno del case.

Sebbene l'efficienza del case dipende dall'implementazione, come regola si può dire che il case va usato quando ci sono delle istruzioni mutuamente esclusive con la stessa probabilità di selezione.

E. L'istruzione di goto

Un'*istruzione* di *goto* è un'istruzione indicante che l'ulteriore elaborazione dev continuare in un'altra parte del programma e precisamente dove c'é l'etichetta indicata.

```
goto    <etichetta>
```

Ogni etichetta - un numero intero di 4 cifre massimo - deve comparire nella sezione di dichiarazione delle etichette prima di essere usata come marca per un'istruzione. L'ambiente di una etichetta L dichiarata in un blocco A è

l'intero testo del blocco A. Questo significa che *una sola* istruzione nella sezione istruzioni di A può avere come marca L:. Qualunque altra istruzione nell'*intero* blocco A può far riferimento ad L in un'istruzione di goto.

esempio (frammento di programma):

```
label    1: { blocco A }
                ...
     procedure    B; { blocco B }
          label    3;
     begin
     3:    writeln ('errore');
                ...
                     goto    3

                     goto    1
     end;    { blocco B }
begin    { blocco A }
1:    writeln ('fallimento prove')
     { nel blocco A non è permesso un goto a 3 }
end.
```

Attenzione: l'effetto di un salto da fuori un'istruzione strutturata a dentro la stessa *non* è definito. Riportiamo tre esempi non corretti; non necessariamente i compilatori indicano errore.

Esempi *scorretti*

```
a)    for    i := 1 to 10 do
           begin    S 1;
           3:    S 2
           end;
      goto    3

b)    if    p then goto 3;
                ...
      if    q then    3: S

c)    procedure    P;
      begin    ...
           3:    S
      end;
      begin    ...
           goto    3
      end.
```

L'istruzione di goto andrebbe riservata solo per quei casi abbastanza fuori dal comune in cui la struttura naturale di un algoritmo deve essere spezzata. È buona regola evitare l'uso di salti per esprimere iterazioni regolari e l'esecuzione condizionata di istruzioni; i salti incrinano infatti l'immagine della struttura dell'elaborazione, rappresentata nella struttura testuale (statica) del programma. Inoltre la mancanza di corrispondenza tra struttura testuale e computazionale (statico e dinamico) è estramemente nociva per la chiarezza del programma e rende molto più difficoltosa la verifica. La presenza di goto in un programma Pascal è spesso il sintomo che il programmatore non ha imparato a "pensare in Pascal" e che è ancora legato ad altri linguaggi dove questo costrutto è necessario.

CAPITOLO 5

TIPI SCALARI E SOTTOCAMPI

A. Tipi scalari

I tipi di dati base in Pascal sono i *tipi scalari*. La loro definizione, costituita da un elenco di identificatori denotanti dei valori, permette la costruzione di un insieme ordinato di valori.

```
type   <identificatore tipo> = ( <identificatore>
       {, <identificatore> } );
```

esempio:

```
type   colore = (bianco, rosso, blu, giallo, porpora, verde, arancio,
                 nero);
       sesso = (maschio, femmina);
       giorno = (lun, mar, mer, gio, ven, sab, dom);
       operatori = (piu, meno, per, diviso);
```

esempio non corretto:

```
type   giornolavorativo = (lun, mar, mer, gio, ven, sab);
       libero = (sab, dom);
```

(c'é un'ambiguità per sab)

Il lettore conosce già il tipo standard booleano definito come:

```
type   boolean = (false, true)
```

Questa definizione introduce gli identificatori standard false e true e specifica che false<true.

Gli operatori di redazione =, <>, <=, >=, e >, sono applicabili a tutti i tipi scalari purché entrambi gli operandi siano dello stesso tipo. L'ordine è determinato dalla sequenza con cui sono elencate le costanti.

Funzioni standard con argomento di tipo scalare sono:

succ(x)	p.e.	succ(blu) = giallo	successore di x
pred(x)		pred(blu) = rosso	antecedente di x
ord(x)		ord(blu) = 2	numero ordinale di x

Il numero ordinale della prima costante elencata è 0,ord(x) = ord-(pred(x))+1.

Supponendo che c e c 1 sono di tipo colore (vedi sopra), b è di tipo booleano, e s 1...sn sono istruzioni arbitrarie, allora le istruzioni che seguono sono significative

```
for  c := nero downto rosso do s1

while  (c1 <> c) and b do s1

if  c>bianco then c := pred (c)

case c of,
        rosso, blu, giallo : s1;
        porpora: s2;
        verde, arancio: s3;
        bianco, nero: s4
end
```

B. Tipo sottocampo

Ad un tipo scalare è associabile un tipo detto sottocampo nella cui definizione basta, semplicemente, indicare il valore più piccolo e quello più grande; il limite inferiore non può essere maggiore del limite superiore. Non è permesso un sottocampo di un tipo reale.

```
type   <identificatore tipo > = <costante> .. <costante>;
```

Dal punto di vista del significato un sottocampo può essere considerato una sostituzione, tagliata su misura, del tipo scalare associato. La validità delle operazioni che coinvolgono dei valori di tipo sottocampo è comunque sempre determinata dallo scalare associato. Considerate, ad esempio, le dichiarazioni:

```
var   a: 1..10; b: 0..30; c: 20..30
```

in cui lo scalare associato ad a, b, c, è intero.

Gli assegnamenti:

```
a := b; c := b; b := c
```

sono tutti validi, anche se la loro esecuzione può, in certi casi, non essere fattibile. Sia nel manuale che nel rapporto, quando si ha a che fare con scarlari, sarà sempre sottointesa la frase : "o un sottocampo di quello".

esempio:

```
type  giorni = (lun, mar, mer, gio, ven, sab, dom); {tipo scalare}
      giolav = lun..ven {sottocampo di giorni}
      indice = 0..63 {sottocampo di intero}
      lettera = 'a'..'z' {sottocampo di car}
```

I tipi sottocampo permettono di rendere più esplicativo il testo del programma. All'implementatore danno l'opportunità di ottimizzare l'uso della memoria e di introdurre prove di validità degli assegnamenti durante l'esecuzione. (Si veda il programma 6.3 per un esempio con tipi sottocampo).

CAPITOLO 6

GENERALITA' SUI TIPI STRUTTURATI I VETTORI IN PARTICOLARE

Scalari e sottocampo sono dei tipi non strutturati; gli altri tipi, in Pascal, sono detti *strutturati*. Come le istruzioni strutturate sono composte da altre istruzioni, così i tipi strutturati sono compsti da altri tipi. Un tipo strutturato è caratterizzato dal tipo dei suoi *componenti* e, cosa molto più importante, dal metodo di strutturazione.

Ogni metodo di strutturazione dispone di un'opzione per indicare la rappresentazione interna preferita. Una definizione di un tipo che ha come prefisso **packed** informa il compilatore che si desidera un'economizzazione di memoria, anche a spese di un aumento del tempo di esecuzione e di una possibile espansione del codice, dovuti alla necessità di impaccare e disimpaccare i dati. È responsabilità dell'utente valutare il peso del baratto efficienza - spazio. In realtà, l'efficienza ed il risparmio di memoria potrebbero anche essere nulli.

Il tipo vettore

Un vettore è costituito da un numero fisso di componenti (definiti quando è introdotto il vettore stesso), tutti dello stesso tipo, chiamati componenti o *tipo base*. Con il nome di una variabile di tipo vettore seguito da un *indice*, posto tra parentesi quadre, si può denotare ed accedere direttamente ad ogni componente. Il tempo necessario per selezionare (accedere) un componente non dipende dal valore del selettore (indice) ed è per questo che i vettori sono chiamati *strutture ad accesso casuale*. Gli indici dei vettori sono computabili e possono essere di tipo sottocampo o scalari - eccettuati interi e reali.

La definizione di un vettore specifica sia il tipo del componente, che può essere qualsiasi, che quello dell'indice. La forma è:

```
type    A = array [T 1] of T 2;
```

dove A è l'identificatore del nuovo tipo; T 1 e il tipo dell'indice; T 2 il tipo dei componenti. Come già detto, mentre T 2 può essere di qualunque tipo, T 1, pur dovendo essere scalare non può essere né reale né intero.

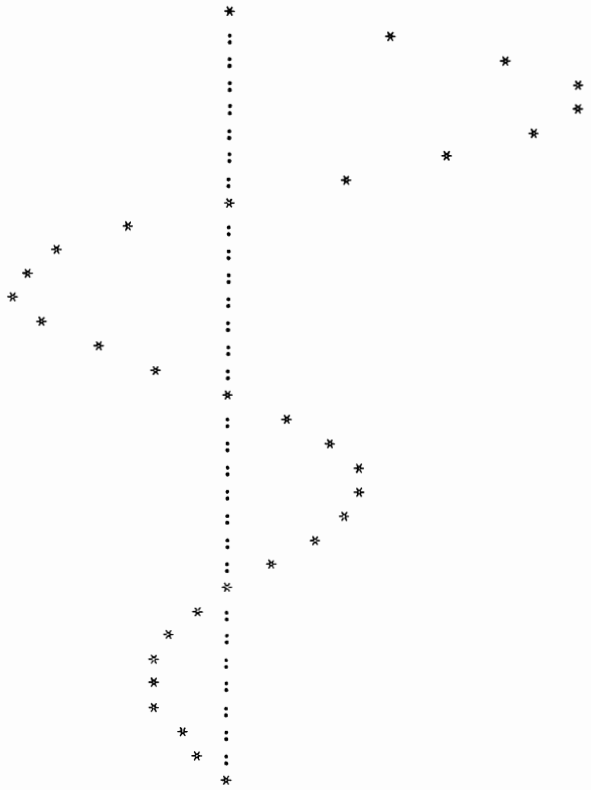
esempi di dichiarazioni di variabili - e - assegnamenti campioni

```
memoria   : array [0..mass] of integer     memoria [ i+j ] := x
malato    : array [giorni] of boolean      malato [ lun ] := true
```

Naturalmente questi esempi presuppongono la definizione di identificatori ausiliari.

```
{ programma 6.1
  dato un elenco, trovare il numero più piccolo e quello più grande }

program    minmass (input, output);

const    n = 20;
var    i, u, v, min, mass, : integer;
       a: array [1..n] of integer;

begin
    { si supponga che a questo punto del programma il vettore contenga i
      valori: 35 68 94 7 88 -5 -3 12 35 9 -6 3 0 -2 74 88 52 43 5 4 }

     min := a[1]; mass := min; i := 2;
     while    i < n do
     begin    u := a[i]; v := a[i+1];
        if    u>v then
        begin    if u>mass then mass := u;
                 if v<min  then min  := v;
        end    else
        begin    if v>mass then mass := v;
                 if u<min then min  := u;
        end
        i := i+2
     end;
     if    i=n then
          if    a[n]>mass then mass := a[n]
          else    if a[n] <min then min := a[n];
     writeln (min, mass)
end.

        94          -6
```

```
{ programma 6.2
  estende il programma 4.9 stampando l'asse x }

program    grafico 2 (output);
const    d = 0.0625; {1/16, 16 linee per intervallo [ x,x+1 ] }
         s = 32; {32 caratteri di larg. per intervallo [ y,y+1 ] }
         hl = 34; {posizione asse x};
         h2 = 68; {larg. linea}
         c = 6.28318; {2*pi} lim = 32;
var    i, j, k, n : integer; x, y, : real;
       a: array [1..h2] of char;
begin for    j := 1 to h2 do a[j] := ' ';
    for    i := 0 to lim do
    begin    x := d*i; y := exp(-x)*sin(c*x);
         a[hl] := ':'; n := round(s*y)+hl; a[n] := '*';
         if    n < hl then k := hl else k := n;
         for    j := 1 to k do write (a [j ] );
         writeln: a [ n ] := ' '
    end
end.
```



(Si consideri come si potrebbe estendere il programma 6.2 per stampare più di una funzione, con o senza l'uso di un vettore)

Poiché T2 può essere di qualsiasi tipo, i componenti di un vettore possono essere strutturati. In particolare se anche T2 è un vettore si parla di vettore *multidimensionale*. La dichiarazione di un vettore multidimensionale M può essere formulata come segue:

```
var   M : array [a..b] of array [c..d] of T;
```

quindi M[i] [j] denota il j-esimo componente (di tipo T) dell'i-esimo componente di M.

Per i vettori ultidimensionali è usuale semplificare la scrittura nel modo seguente:

```
var   M : array [a..b, c..d] of T;                    T;
```

e

```
M[i, j]
```

Si può considerare M come matrice e dire che M[i, j] è il j-esimo componente (riga j) dell'i-esimo componente di M (colonna i).

La definizione non è limitata a due dimensioni in quanto T può essere a sua volta strutturato. In generale la forma abbreviata é:

```
type   <identificatore tipo> =
       array [ <tipo indice> { ' <tipo indice>} ] of <tipo componente>;
```

Se sono specificati n tipi-indice si dice che il vettore è a *n dimensioni*; un componente è individuato da espressioni con n indici.

```
{ programma 6.3
  moltiplicazione di matrici }


program      moltmatr (input, output);

const   m = 4; p = 3; n = 2;
var     i : 1..m; j : 1..n; k : 1..p;
        s : integer;
        a : array [ 1..m, 1..p ] of integer;
        b : array [ 1..p, 1..n ] of integer;
        c : array [ 1..m. 1..n ] of integer;
```

```
begin    {assegna i valori iniziali ad a e b}
    for    i := 1 to m do
    begin for    k := 1 to p do
          begin    read(s); write(s); a [ i,k ] := s
          end;
          writeln


    end;
    writeln;
    for    k := 1 to p do
    begin for    j := 1 to n do
          begin    read(s); write(s); b[k,j] := s
          end;
          writeln

    end;
    writeln;
    { moltiplicazione a*b }
    for    i := 1 to m do
    begin for    j := 1 to n do
          begin    s := 0;
             for    k := 1 to p do s := s+a [i,k]*b[k,j];
             c[i,j] := s; write(s)
          end;
          writeln
    end;
    writeln
end.
```

```
 1        2        3
-2        0        2
 1        0        1
-1        2       -3

-1        3
-2        2
 2        1

 1       10
 6       -4
 1        4
-9       -2
```

Le *stringhe* sono state, precedentemente, definite come caratteri racchiusi in

apici (capitolo 1). Le stringhe costituite da un singolo carattere sono costanti del tipo standard char (capitolo 2); quelle costituite da n caratteri (n>1), sono definite come costanti del tipo:

```
packed array [1..n] of char
```

L'assegnamento è possibile fra vettori se sono di *identico* tipo. Gli operatori di relazione =, <>, <, >,<=, e >= sono applicabili a vettori identici di caratteri impaccati e l'ordinamento è determinato dall'insieme di caratteri base.

L'accesso ad un unico elemento di un vettore impaccato è spesso costoso; si consiglia il programmatore di impaccare o disimpiccare un vettore impaccato con un'unica operazione. Sia A una variabile vettore di tipo:

```
array    [m..n] of T
```

e Z una variabile di tipo

```
packed array    [u..v] of T
```

dove (n-m) >= (v-u) allora

```
impacca(A,i,Z)        significa    for   j := u to v do
                                      Z[j] := A[j-u+i]
disimpacca(Z,A,i)     significa    for   j := u to v do
                                      A[j-u+i] := Z[j]
```

dove j denota una variabile con non ricorre in altri punti del programma.

CAPITOLO 7

TIPI RECORD

Il tipo record è forse il più flessibile dei tipi di dato. Concettualmente un record è uno schema per una struttura le cui parti possono avere caratteristiche molto diverse. Supponiamo, ad esempio, di voler memorizzare informazioni su una persona: nome, numero codice fiscale, sesso, data di nascita, numero persone a carico e stato civile. Inoltre, se la persona è sposata o vedova, viene riportata la data di matrimonio (l'ultimo); se divorziata, la data del più recente divorzio e se questo è il primo o no; se scapolo (nubile), se abita da solo /a.
Tutte queste informazioni possono essere raggruppate in un "record".

Il modo più formale si può dire che un record è una struttura costituita da un numero fisso di componenti chiamati *campi*.
Diversamente dai componenti di un vettore, i campi non devono necessariamente essere dello stesso tipo e non possono essere individuati con indici. La definizione del record specifica per ogni componente il suo tipo ed un identificatore che lo individua: *identificatore di campo*. L'ambiente di un identificatore di campo è il record più interno nel quale è definito. Affinché il tipo dei componenti selezionati siano evidenti dalla lettura del testo (senza l'esecuzione del programma), il selettore è costituito da identificatori di campo costanti: non sono cioé permessi valori calcolabili.

Come esempio si supponga di voler fare delle elaborazioni con numeri complessi della forma a + ib, dove a e b sono numeri reali e i è la radice quadrata di -1. Pur non esistendo un tipo standard "complesso", il programmatore può facilmente definire un record per rappresentare i numeri complessi. Il record avrà bisogno di due campi, uno per la parte reale e l'altro per la parte immaginaria, entrambi di tipo reale. Sintatticamente il tipo record è:

```
<tipo record>     := record <elenco campi> end
<elenco campi>    := <parte fissa> | <parte fissa> ; <parte variabile> |
                     <parte variabile>
<parte fissa>     := <sezione record> {; <sezione record> }
<sezione record>  := <identificatore campo>
                     {, <identificatore campo> }: <tipo> | <vuoto>
```

Applicando queste regole, si possono scrivere la definizione e la dichiarazione che seguono:

```
type   complesso = record re,im : real
                   end;
var    x : complesso;
```

dove complesso è l'identificatore del tipo, re e im sono gli identificatori dei campi e x è una variabile di tipo complesso. x è quindi un record costituito da due componenti -o campi.

Allo stesso modo una variabile rappresentante una data può essere definita come:

```
data = record mese: (gen, feb, apr, mag, giu, lug, ago, set, ott, nov, dic);
              giorno: 1..31;
              anno: integer
       end
```

un gioco come:

```
gioco = record gen: (palla, trottola, barca, bambola, costruzioni,
                     modelli, libri);
               costo: real;
               ricevuto: data;
               interesse: (molto, unpo, poco, nessuno)
        end
```

o un compito a casa come:

```
compito = record soggetto: (storia, lingua, lett, matem, fisica, scienze);
                 assegnato: data;
                 tiposc: 1..4;
                 voto: 0..10
          end
```

Il nome del record seguito da un punto a sua volta seguito dall'identificatore di campo permette di far riferimento ad un componente. Per assegnare 5+3i alla variabile x precedentemente dichiarata si scrive:

```
x.re := 5;
x.im := 3;
```

Se il record è all'interno di un'altra struttura, la chiamata del record rifletterà

quella struttura. Si voglia, per esempio, memorizzare la recente vaccinazione antivaiolosa per ogni membro di una famiglia. Una delle possibilità è quella di definire i membri come degli scalari e la data in un vettore di record:

```
type   famiglia= (padre, madre, bimb1, bimb2, bimb3)
var    vaccinaz. : array [famiglia] of data
```

un aggiornamento potrebbe essere del tipo:

```
vaccinaz [bimb3] .mese   := apr;
vaccinaz [bimb3] .giorno := 23;
vaccinaz [bimb3] .anno   := 1973
```

Si noti che il tipo "data" include anche, per esempio, il 31 aprile.

```
{ programma 7.1
  operazione sui numeri complessi }

program   complessi (output);
const fac = 4;

type   comples = record re,im : integer end;
var    x,y : comples;
       n : integer;
begin
    x.re := 2; x.im := 7;
    y.re := 6; y.im := 3;
    for    n := 1 to 4 do
    begin
        writeln ('x =', x.re:3,x.im:3, 'y = ', y,re:3,y.im:3);
        {x + y}
        writlen ('somma =', x.re+y.re:3,
        {x*y}                x.im+y.im:3);
        writeln ('prodotto =' ,x.re*y.re - x.im*y.im:3),
                               x.re*y.im + x.im*y.re:3);
        writeln;
        x.re := x.re + fac; x.im := x.im - fac;
    end
end.
```

```
x =    2  7    y =    6  3
somma =    8  10
prodotto =    -9  48

x =    6  3    y =    6  3
somma =    12  6
prodotto =    27  36

x =    10  -1    y =    6  3
somma =    16  2
prodotto =    63  24

X =    14  -5    y =    6  3
somma =    20  -2
prodotto =    99  12
```

La sintassi dei tipi record permette anche di costruire records con *parti varianti*: un record può cioé essere specificato come costituito da parecchie *varianti*. Questo significa che variabili diverse, sebbene definite dello stesso tipo, possono avere strutture che differiscono e queste differenze possono essere sia un numero diverso di componenti che componenti di tipo diverso.

Ciascuna variante è caratterizzata da un elenco, tra parentesi, di dichiarazione dei suoi componenti possibili. Ciascun elenco è etichettato con una o più etichette mentre l'insieme degli elenchi è preceduto da una proposizione di selezione che specifica il tipo di dato delle etichette. In sostanza, si specificano i tipi che permettono di discriminare le varianti.
Per chiarire quanto detto si supponga di avere un:

```
type    statocivile = (sposato, vedovo, divorziato, celnu)
```

Si possono quindi descrivere le persone con dati del tipo:

```
type    persona =
    record    <attributi comuni a tutte le persone>;
        case    statocivile    of
        sposato: ( <solo campi sposati> );
        celnu: ( <solo campi celibe/nubile> );

    end
```

Di solito un campo del record stesso indica la variante valida in quel momento.

È, ad esmpio, verosimile che il record persona definito sopra contenga un campo comune

sta : statocivile

Questa frequente situazione permette di abbreviare la scrittura includendo la dichiarazione del campo selezionato - campo *discriminatore* nel case stesso; si scrive quindi:

case stat : statocivile **of**

La sintassi che definisce la parte variante è:

<parte variante> := **case** <campo discriminatore>
<identificatore tipo> **of** <variante>
{; <variante> }
<variante>: = <elenco etichette case> : (<elenco campi>) | <vuoto>
<elenco etichette case>: = <etichetta case> {, <etichetta case> }
<etichetta case>: = <costante>
<campo discriminatore>: = <identificatore> : | <vuoto>

Volendo caratterizzare una persona con un record con variante, è utile, dapprima, abbozzarne le informazioni:

I. Persona
- A . nome (nome, cognome)
- B . codice fiscale (stringa alfanumerica)
- C . sesso (maschio, femmina)
- D . data di nascita (mese, giorno, anno)
- E . numero persone a carico (intero)
- F . stato civile
  - se sposato o vedovo
    - a. data del matrimonio (mese, giorno, anno)
  - se divorziato
    - a. data del divorzio (mese, giorno, anno)
    - b. primo divorzio (vero, falso)
  - se celibe/nubile
    - a. residenza indipendente (vero, falso)

In figura 7.a sono rappresentati due campioni con diversi attributi.

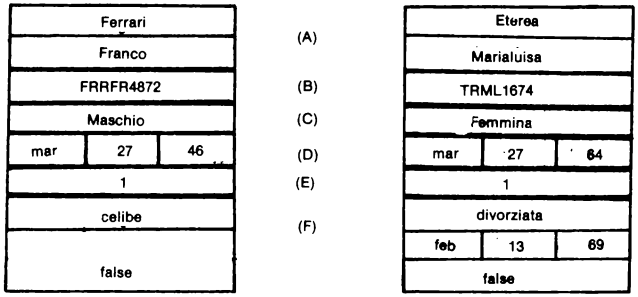


Fig. 7.a - Due persone campione.

Un record che definisce una persona può essere definito come:

```
type    alfa = packed array [1..10] of char;
        statocivile = (sposato, vedovo, divorziato, celnu)
        data = record mese : (gen, feb, mar, apr, mag, giu, lug, ago, set,
                              ott, nov, dic);
                      giorno : 1..31
                      anno   : integer
                  end;
        persona = record
                      nom : record nome, cognome : alfa
                                end;
                      cod  : packed array [1..13] of char;
                      ses  : (maschio, femmina);
                      nas  : data;
                      car  : integer;
                      case   stat : statocivile of
                          sposato, vedovo: (datma:data);
                          divorziato: (datdi:data; primodiv:boolean);
                          celnu:      (indip : boolean)
                  end; {persona}
```

Attenzione:

1. Tutti i nomi dei campi devono essere distinti anche se sono in varianti diverse.
2. Se l'elenco dei campi per una etichetta L è vuoto, la forma è: L : ().
3. L'elenco dei campi può avere solo una parte variante e deve seguire le parti fisse; le parti varianti possono nidificare.

Per esemplificare la semplicità e linearità dei riferimenti ai componenti di un

record riportiamo gli assegnamenti che permettono di creare un record di tipo persona (p è la variabile).

```
p.nom.cognome := 'eterea';
p.nom.nome := 'marialuisa';
p.cod := 'TRML167490A12';
p.ses := femmina;
p.nas.mese := mar,
p.nas.giorno := 27;
p.nas.anno := 1964;
p.car := 0;
p.stat := celnubile;
p.indip := false
```

A. L'istruzione with

La notazione riportata sopra, che in certi casi può divenire lunga e noiosa, è abbreviabile con l'istruzione **with**. Con with si va ad aprire l'ambiente contenente l'identificatore di campo della variabile record specificata, consentendo l'uso dell'identificatore di campo come identificatore di variabile. Questa direttiva fornisce al compilatore l'opportunità di ottimizzare l'istruzione associata. La forma è:

with <variabile record> {, <variabile record> } **do** <istruzione>

Nelle istruzioni interne all'istruzione with, un campo di una variabile di tipo record è denotato solamente dall'identificatore di campo - non lo si fa cioé precedere dalla notazione di tutta la variabile record.

La direttiva with riportata sotto, rende gli assegnamenti equivalenti a quelli della sezione precedente:

```
with    p'nom'nas do
begin
   cognome  := 'eterea';
   nome     := 'marialuisa';
   cod      := 'TRML167490A12';
   ses      := femmina';
   mese     := mar;
   giorno   := 27;
   anno     := 1964;
   car      := 0;
   stat     := celnub;
   indip    := false
end    {with}
```

Allo stesso modo:

```
var   datacorrente : data;
          ...
      with   datacorrente do
          if   mese=dic then
               begin   mese := gen; anno := anno+1
               end
          else   mese := succ(mese)
```

è equivalente a

```
var   datacorrente : data;
          ...
      if   datacorrente.mese = dic then
           begin   datacorrente.mese := gen;
                   datacorrente.anno := datacorrente.anno+1
           end
      else   datacorrente.mese   succ(datacorrente.mese)
```

L'istruzione seguente aggiorna la vaccinazione che si è vista in un esempio precedente:

```
with   vaccinaz [bimb3] do
    begin   mese := apr; giorno := 23; anno := 1973
    end
```

Nessun elemento dell'elenco delle variabili associate al with può essere modificato con assegnamenti nelle istruzioni subordinate al do del with. Vale a dire, dato:

```
with   r do S
```

r non può contenere variabili modificate da S; per esempio:

```
with   a [i] do
    begin   ...
        i := i+1
    end
```

Non è concesso.

La forma:

```
with   r1, r2,...,rn do S
```

è equivalente a

```
with   r1 do
    with   r2 do

                with rn do S
```

Mentre:

```
var   a : array [2..8] of integer;
      a : 2..8;
```

NON è permesso, in quanto la definizione di a è ambigua,

```
var   a : integer;
      b : record a :real; b : boolean
          end;
```

È permesso in quanto l'intero a è facilmente distinguibile dal reale "b.a". Allo stesso modo la variabile record b è distinguibile dal booleano "b.b". All'interno dell'istruzione S in

```
with   b do S
```

gli identificatori a e b denotano rispettivamente i componenti b.a e b.b.

CAPITOLO 8

TIPI SET

Un tipo set definisce l'insieme dei valori che è l'insieme potenza del suo tipo base: definisce cioé l'insieme di tutti i sottoinsiemi di valori del tipo base, insieme vuoto incluso. Il tipo base deve essere scalare o sottocampo

type <identificatore> = **set of** <tipobase>;

Le implementazioni di Pascal possono definire dei limiti, anche molto piccoli (p.e. il numero di bits di una parola), per le dimensioni degli insiemi.

Gli insiemi sono costruiti a partire dai loro elementi per mezzo di costruttori di insiemi che sono denotati da <set> nella sintassi. Consistono nell'enumerazione degli elementi dell'insieme; più precisamente, in espressioni del tipo base, separate da virgole e racchiuse da parentesi d'insieme [e]. [] denotano l'insieme vuoto.

<set> ::= [<elenco elementi>]
<elenco elementi> ::= <elemento> {, <elemento> } | <vuoto>
<elemento> ::= <espressione> | <espressione>..<espressione>

La forma m..n denota l'insieme di tutti gli elementi i del tipo base tali che m<=i<=n. Se m>n, [m..n] denota l'insieme vuoto.

Esempi di costruttori di insiemi:
[13]
[i+j, i-j]
['A' .. 'Z', '0' .. '9']

I seguenti operatori sono applicabili su tutti gli oggetti con struttura set:

+ unione
* intersezione
- differenza (p.e. A-B denota l'insieme di tutti gli elementi di A che non sono anche elementi di B).

Gli operatori relazionali applicabili ad operandi di tipo set sono:

= <>	verifica su uguaglianza/diseguaglianza
<= >=	verifica su inclusione
in	appartenenza. Il secondo operando è di tipo set mentre il primo è del tipo del 'tipo base associato'; il risultato è vero quando il primo è un elemento del secondo, altrimenti è falso.

esempi di dichiarazioni -e- assegnamenti

```
type  primario := (rosso, giallo, blu);   hue 1 := [rosso]; hue 2 := [];
      colore = set of primario;           hue 2 := hue 2 + [succ (rosso)]
var   hue 1, hue 2 : colore;
var   ch : char;                          chset 1:= ['d', 'a', 'g'];
      chset 1, chset 2: set of 'a'..'z';   chset 2 := ['a'..'z'] - [ch]
var   codop: set of 0..7;                 add:= [2,3] <= codop
      add: boolean;
```

Le operazioni sui set sono relativamente veloci e possono essere usate per eliminare verifiche più complicate. Una forma di verifica più semplice per:

```
if   (ch= 'a') or (ch= 'b') or (ch= 'c') or (ch= 'd') or (ch= 'z') then s
```

è:

```
if   ch in ['a'..'d', 'z'] then s
```

```
{ programma 8.1
  esempio di operazione sui sets }

program   setop (output);

type   giorni = (Lu, ma, me, gi, ve, sa, dom);
       settimane = set of giorni;
var    sett, lavoro, libero : settimane;
       d : giorni;

procedure   check (s:settimane); {procedura introdotta nel capitolo 11}
    var   d:giorni;
begin   write (' ')
    for   d := lu to do
          if d in s then ('x') else write ('o');
    writeln
end;   {check}
```

```
begin   lavoro := []; libero := [];
   sett := [lu..dom];
   d := sa; libero := [d] + [dom] + libero;
   check (libero);
   Lavoro := sett - libero; check (lavoro);
   if libero <= sett then write ('0');
   if sett >= lavoro then write ('K');
   if not (lavoro >= libero) then write ('aldo');
   if [sa] <= lavoro then write ('dimentica');
   writeln
end.
```

```
00000XX
XXXXX00
OK aldo
```

Considerazione sullo sviluppo dei programmi

La programmazione - intesa come progetto e formulazione di algoritmi - è generalmente un'attività complicata che richiede padronanza di tecniche specifiche e continua considerazione di numerosi dettagli. Solo in casi eccezionali ci sarà un'unica buona soluzione. Solitamente, ci sono tali e tante soluzioni che la scelta di un programma ottimale richiede una completa analisi non solo degli algoritmi ed elaboratori disponibili ma anche del modo in cui il programma sarà usato più spesso.

La costruzione di un algoritmo dovrebbe, quindi, consistere in una sequenza di deliberazioni, investigazioni e decisioni di progetto. Nei primi stadi è preferibile concentrare l'attenzione sul problema nella sua globalità ed abbozzare una soluzione senza dar troppa importanza, ai dettagli. Con il progredire dell'attività di progettazione il problema può essere suddiviso in sottoproblemi e gradualmente si considerano maggiormente i dettagli delle specifiche del problema e le caratteristiche degli strumenti disponibili. Due locuzioni caratterizzano questo modo di affrontare la programmazione: *raffinamenti successivi* [2] e *programmazione strutturata* [4].

Il resto di questo capitolo mostra lo sviluppo di un algoritmo riformulando un esempio che C.A.R. Hoare presenta in *structured programming* [4], "Nota sull'organizzazione dei dati": la riformulazione è necessaria per renderlo corretto rispetto alla notazione Pascal.

Si tratta di trovare i numeri primi appartenenti al campo 2..n, con n>=2. Dopo un confronto dei vari algoritmi, si è scelto, a causa della sua semplicità -né moltiplicazioni né divisioni quello del setaccio di Eratostene.

La prima formulazione è verbale.

1. Metti tutti i numeri tra 2 e n nel "setaccio".
2. Scegli e rimuovi il numero più piccolo rimasto nel setaccio.
3. Includi questo numero nei "numeri primi"
4. Rimuovi dal setaccio tutti i multipli di questo numero
5. Se il setaccio non è vuoto ripeti i passi 2-5.

Sebbene l'inizializzazione delle variabili è il primo passo nell'esecuzione di un programma, spesso è l'ultimo nello sviluppo dello stesso. Una completa comprensione di un algoritmo è il prerequisito per compiere delle inizializzazioni appropriate; un aggiornamento di queste inizializzazioni con ogni modifica del programma è necessario per mantenere il programma in grado di girare. (Sfortunatamente, l'aggiornamento non è sempre sufficiente!).

Hoara ha scelto un tipo set con gli elementi 2..n per rappresentare sia il setaccio che i numeri primi. Il programma che segue è una leggera variazione dell'abbozzo da lui presentato.

```
const   n = 10000;
var     setac, primi : set of 2..n;
        prossimo, j : integer;
begin   {inizializzazione}
    setac := [2..n]; primi := []; prossimo := 2;
    repeat   {cerca il prossimo num. primo}
        while not (prossimo in setac) do prossimo := succ (prossimo);
        primi := primi + [prossimo];
        j := prossimo;
        while   j<=n do {eliminazione}
            begin   setac := setac - [j]; j := j+prossimo
            end
    until   setac = []
end.
```

Come esercizio Hoara propone di riscrivere il programma, cosicché gli insiemi rappresentano solo i numeri dispari. Il programma che segue è una proposta: si noti la stretta correlazione con la prima soluzione.

```
const    n = 5000; {n' = n div 2}
var    setac, primi : set of 2..n;
       prossimo, j, c, : integer;
begin    {inizializ.}
     setac := [ 2..n ]; primi := []; prossimo := 2;
     repeat    {cerca il prossimo numero primo}
          while not    (prossimo in setac) do prossimo := succ(prossimo);
          primi := primi + [prossimo];
          c := 2*prossimo - 1; {c=nuovo primo}
          j := prossimo;
          while    j<=n do {eliminazione}
               begin    setac := setac - [j]; j := j+c
               end
     until    setac = []
end.
```

È desiderabile che tutte le operazioni base sugli insiemi siano relativamente veloci. Parecchie implementazioni restringono l'ampiezza massima degli insiemi alla lunghezza della loro "parola", di modo che ogni elemento dell'insieme base è rappresentato da un bit (O significa assenza, 1 presenza). La maggior parte delle implementazioni non accetterebbe perciò un insieme con 10,000 elementi. Queste considerazioni portano ad un riaggiustamento della rappresentazione dei dati, come è mostrato nel programma 8.2.

Un insieme con molti elementi può essere rappresentato con un vettore di insiemi più piccoli cosicché ciascuno "entra" in una parola (dipendente dalla implementazione). Il programma seguente usa il secondo abbozzo come modello astratto dell'algoritmo. Il setaccio ed i numeri primi sono ridefiniti come vettori di insiemi; prossimo è definito come record; L'output come record: L'output non è sviluppato.

```
{ programma 8.2
  genera i numeri primi tra 3 e 10000 usando un setaccio contenente interi
  dispari }

program    prumprimi (output);

const    Lungpar = 59; {dipendente dell'implementazione}
         maxbit = 58;
         w = 84; {w = n div Lungpar div 2}
var    setac, primi : array [o..w] of set of 0..maxbit;
       prossimo : record parola, bit : integer
                    end;
       j, k, t, c : integer; vuoto : boolean;
begin    {iniz.}
    for    t := o to w do
         begin    setac [t] := [0..maxbit]; primi [t] := [] end;
    setac [0] := setac [0] - [0]; prossimo.parola := 0;
    prossimo.bit := 1; vuoto := false;
    with    prossimo do
    repeat    {cerca prossimo num.primo}
         while not    (bit in setac [parola]) do bit := succ(bit);
         primi [parola] := primi[parola] + [bit];
         c := 2*bit + 1;
         j := bit; k := parola;
         while    k<= do {elimin.}
         begin    setac [k] := setac [j];
              k := k + parola*2; j := j + c;
              while    j>maxbit do
                   begin    k := k + 1; j := j - lungpar
                   end;
         end;
         if    setac [parola] = []  then
              begin    vuoto := true; bit := 0
              end;
         while    vuoto and (parola <w) do
              begin    parola := parola+1; vuoto := setac [parola] = []
              end
    until    vuoto; {fine with}
end.
```

CAPITOLO 9

TIPI FLUSSO (FILE)

In molti casi il metodo di strutturazione più semplice è la sequenza. Nel campo dell'elaborazione dati i termini generalmente accettati per descrivere una sequenza sono *flusso sequenziale* (sequential file). Pascal usa la parola **file** per specificare una struttura costituita da una sequenza di componenti tutti dello stesso tipo.

Nella sequenza è definito un ordinamento naturale dei componenti ed in ogni istante solo un componente è direttamente accessibile. Si può accedere agli altri componenti progredendo sequenzialmente lungo il flusso. Il numero dei componenti - *lunghezza* del flusso non è fissato dalla definizione del tipo: questa è una caratteristica che distingue nettamente un flusso da un vettore. Un flusso senza componenti è detto *vuoto*.

type <identificatore> = **file of** <tipo>;

La dichiarazione di ogni variabile flusso f automaticamente introduce una *variabile buffer*, denotata da f↑, del tipo componente. Può essere considerata come una *finestra* attraverso la quale si può sia ispezionare (leggere) un componente esistente sia appendere (scrivere) nuovi componenti; questa finestra è mossa automaticamente da certi operatori di flusso.

L'elaborazione sequenziale e l'esistenza di un "buffer" fa pensare che i flussi possono essere associati con *memorie secondarie* e *periferiche*. Il modo preciso in cui i componenti sono allocati è dipendente dall'implementazione, ma noi facciamo l'assunzione che solo alcuni dei componenti sono presenti nella memoria principale in un certo instante, e che solo il componente indicato con f↑ è direttamente accessibile.

Quando la finestra f↑ è messa oltre la fine del flusso f (**end of file**), la funzione booleana standard eof (f) ritorna il valore true, negli altri casi ritorna il valore false. Gli operatori base per la manipolazione dei flussi sono:

reset (f) riposiziona la finestra del file all'inizio per permettere la lettura. Assegna cioé a f↑ il valore del primo elemento di f; eof (f) diviene false se f non è vuoto; nel caso in cui f sia vuoto f↑ non è definito e eof (f) rimane true.

rewrite (f) precede la riscrittura di un flusso f. Il valore corrente di f↑ è sostituito con il flusso vuoto, eof (f) diviene true e può essere scritto un nuovo flusso.

get (f) avanza la finestra sul prossimo componente: assegna cioé il valore di questo componente alla variabile buffer f↑. Se non esiste il prossimo componente eof (f) diviene true, ed il valore risultante di f↑ non è definito. L'effetto di get (f) è definito solo se eof (f) è false prima della sua esecuzione.

put (f) appende il valore della variabile buffer f↑ al flusso f. L'effetto è definito solo se prima dell'esecuzione il predicato eof (f) è vero, eof (f) rimane vero e f↑ diviene indefinito.

In linea teorica tutte le operazioni di generazioni ed ispezione su un flusso sequenziale possono essere espresse per mezzo dei quattro operatori primitivi di flusso ed il predicato eof. In pratica, è spesso naturale combinare l'operazione di avanzamento nel flusso con l'accesso alla variabile buffer. Introduciamo perciò le due procedure di read (leggi) e write (scrivi):

read (f,x) è equivalente a x := f↑; get (f)
write (f,x) è equivalente a f↑ := x; put (f)

Nota: Lo standard definito dal rapporto accenna a queste abbreviazioni solo per x di tipo char.

Il vantaggio di usare queste procedure sta non solo nella brevità, ma anche nella semplicità concettuale, in quanto l'esistenza di una variabile buffer f↑, il cui valore è alle volte indefinito, può essere ignorato. La variabile buffer può tuttavia essere utile come dispositivo per "guardare avanti".

Esempi di dichiarazioni -e-	istruzioni con flussi
var dati : **file of** integer;	a := dati↑; get (dati)
a : integer;	read (dati,a)
var club : **file of** persona;	club↑ := p; put (club)
p : persona	write (club,p)

Esempi di spezzoni di programmi:

1. Legge il flusso f di numeri reali e calcola la somma S.

```
S := 0; reset (f)
while not   eof (f) do
    begin   read (f,x); S := S+x
    end.
```

2. Il seguente pezzo di programma opera su due flussi costituiti da sequenze ordinate di interi

f1,f2,....fm e g1,g2,...,gn

tali che f(i+1) >= f(i) e g(j+1) >= g(j), per tutti gli i,j e li *fonde* in un flusso ordinato h tale che

h(k+1) >= h(k) per k = 1,2,...,(m+n-1).

Fa uso delle seguenti variabili:

```
finefg : boolean;
f,g,h : file of integer

{ spezzone di programma
  fonde f e g in h }

begin   reset (f); reset (g); reset (h);
    finefg := eof(f) or eof(g);
    while not   finefg do
    begin if   f↑ <g ↑ then
        begin   h↑ := f↑; get(f);
            finefg := eof(f)
        end else
        begin h↑ := g↑; get(g);
            finefg := eof(g)
        end;
        put(h)
    end;

    while not eof(g) do
    begin   h↑ := g↑; put(h);
        get(h)
    end;
    while not   eof(f) do
    begin   h↑ := f↑; put(h);
        get(f)
    end
end.
```

I flussi possono essere locali ad un programma (o una procedura), o possono anche esistere esternamente al programma e vengono chiamati *flussi esterni*. I flussi esterni sono passati nel programma come parametri nell'intestazione dello stesso (si veda il capitolo 13).

A. Flussi-testo

I *flussi-testo* (textfiles) sono flussi i cui componenti sono caratteri. Il tipo standard testo è definito come segue:

```
type text = file of char;
```

I testi sono solitamente suddivisi in *linee*. Un metodo semplice per indicare la separazione di due linee consecutive consiste nell'usare dei caratteri di controllo. Per esempio, nell'insieme dei caratteri ASCII i due caratteri *cr* (ritorno carrello) e *lf* (interlinea) sono usati per segnare la fine di una linea. Tuttavia parecchi sistemi usano un insieme di caratteri privo di questi caratteri di controllo; questo implica che devono essere usati altri metodi per indicare la fine linea.

Possiamo indicare il tipo testo come definito sul tipo base char (contenente solo caratteri stampabili) esteso con un (ipotetico) carattere separatore di linee.
Questo carattere di controllo non può essere assegnato ad una variabile di tipo char, ma può essere sia riconosciuto che generato da seguenti operatori speciali per flussi-testo:

writeln (x)	termina la linea corrente del flusso-testo x
readln (x)	salta all'inizio della prossima linea del flusso-testo (x↑ diviene il primo carattere della prossima linea)
eoln (x)	una funzione booleana che indica se è stata raggiunta la fine della linea corrente nel flusso-testo x

Se f è un flusso-testo e h una variabile carattere, la seguente scrittura abbreviata può essere usata al posto degli operatori generali di flusso.

forma abbreviata	forma espansa
write (f,ch)	f↑ := ch; put(f)
read (f,ch)	ch := f↑; get(f)

Gli schemi di programmi che seguono usano le convenzioni riportate sopra per dimostrare alcune tipiche operazioni fatte su flussi-testo.

1. Scrittura di un testo j. Si supponga che P(c) computi un carattere (il prossimo) e lo assegni al parametro c. Se la linea corrente deve essere terminata, una variabile booleana p è posizionata a true; se è il testo a dover essere terminato, q è posizionata true.

```
rewrite(y);
repeat
    repeat    P(c); write(y,c)
    until    p;
    writeln(y)
until    q
```

2. Lettura di un testo x. Si supponga che Q(c) denoti l'elaborazione di un carattere (il prossimo) c. R indica un'azione che deve essere eseguita all'incontro di un fine linea.

```
reset(x);
while not    eof(x)  do
begin
    while not    eoln(x)  do
        begin    read (x, c); Q (c)
        end;
    R; readln;
end.
```

3. Copiatura di un testo x su un testo y, conservando la sequenza delle linee di x.

```
reset(x); rewrite(y);
while not    eof(x)  do
begin    {copia una linea}
    while not    eoln(x) do
        begin    read (x,c); write (y,c)
        end;
    readln(x); writeln(y)
end
```

B. I flussi standard "input" e "output"

I flussi "input" e "output" solitamente rappresentano i dispositivi di I/O standard di un elaboratore (come il lettore di schede o la stampante).
Sono perciò la principale linea di comunicazione tra l'elaboratore e l'utente.

Poiché questi due flussi sono usati molto spesso, sono considerati come"valori per difetto" nelle operazioni su flussi-testo quando il flusso f non è esplicitamente indicato. Vale a dire:

	è equivalente a
write (ch)	write (output, ch)
read (ch)	read (input, ch)
writeln	writeln (output)
readln	readl (input)
eof	eof (input)
eoln	eoln (input)

Nota: Le procedure standard *reset (rewrite)* non devono essere applicate ai flussi input (output).

In accordo con quanto detto, nel caso in cui x è "input" e y è "output", i primi due schemi di programma possono essere riscritti come segue: (si supponga **var** ch: char)

Scrittura di un testo sul flusso "output":

```
repeat
    repeat   P(ch); write(ch)
    until   p;
    writeln
until   q
```

Lettura di un testo del flusso "input";

```
while not   eof do
begin   {elabora una linea}
    while not   eoln do
        begin   read(ch); Q(ch)
        end
    R; readln
end
```

Ulteriori estensioni delle procedure write e read (per una comoda gestione di dati di input/output leggibili) sono descritte nel capitolo 12.

Nei prossimi 2 esempi si mostra l'uso dei flussi-testo di input e output. (Si considerino i cambiamenti che sarebbero necessari se, invece di read e write, si usasse solo get e put)

```
{programma 9.1 -- conteggio frequenza lettere nel flusso input}
program    contfreq (input, output);

var    ch : char;
       cont : array ['a' .. 'z'] of integer;
       lettera : set of 'a' .. 'z';
begin    lettera := ['a' ..'z'];
     for    ch := 'a' to 'z' do count [ch] := 0;
     while not    eof do
     begin
          while not    eoln do
          begin    read(ch); write(ch);
               if (ch) in lettera then count [ch] := cont [ch] + 1
          end;
          writeln; readln
     end
end.
```

In certi sistemi quando un flusso è inviato su una stampante, il primo carattere di ogni linea è usato come carattere di controllo: il primo carattere non è cioé stampato, ma è interpretato per l'avanzamento della carta della stampante. Le seguenti convenzioni sono diffusamente impiegate:

spazio	:	un interlinea prima di stampare
'0'	:	due interlinee prima di stampare
'1'	:	salto all'inizio della prossima pagina prima di stampare
'+'	:	nessuna interlinea (sovrascrittura)

Il programma seguente inserisce uno spazio all'inizio di ogni linea: si ottiene quindi una stampa normale con una interlinea.

```
{programma 9.2 -- inserzione di uno spazio guida}

program    inserzione (input, output);
var    ch : char;
begin
     while not eof do
     begin    write (' ');
          while not    eoln do
               begin    read(ch); write(ch)
               end;
          writeln; readln
     end
end.
```

Se le procedure di read e write sono usate senza l'indicazione del parametro flusso, per difetto, sono assunti i flussi input e output; essi *devono* però essere elencati nella lista dei parametri nell'intestazione del programma.

CAPITOLO 10

TIPI PUNTATORE

Una *variabile statica* (allocata staticamente) è una variabile che è dichiarata in un programma e che individuabile per mezzo di un identificatore. È chiamata statica, in quanto esiste - le è, cioé, riservata della memoria - durante tutta l'esecuzione del blocco a cui appartiene. Una variabile può, però, essere generata dinamicamente - senza alcuna correlazione con la struttura statica del programma - con la procedura **new**. Una tale variabile è chiamata *variabile dinamica.*

Le variabili dinamiche non sono dichiarate esplicitamente e non possono essere chiamate direttamente con degli identificatori. Per generare una variabile è necessaio un *puntatore* (pointer) che non è niente altro che l'indirizzo di memoria della variabile appena allocata. Un tipo puntatore P è perciò costituito da un insieme di valori che puntano ad elementi di un certo tipo T; P viene detto *legato* a T. Il valore **nil** è un elemento sempre presente in P e non punta ad alcun elemento.

tipe <identificatore> = ↑ <identificatore tipo>:

Se, per esempio, p è una variabile puntatore legata ad un tipo T, la dichiarazione

var p= ↑T

crea p come riferimento ad una variabile di tipo T, e p↑ denota quella variabile. Per creare o allocare una tale variabile è usata la procedura standard *new*. La chiamata new(p) alloca una variabile di tipo T ed assegna il suo indirizzo a p.

I puntatori sono strumento semplice per la costruzione di strutture dati complicate e flessibili. Se il tipo T è una struttura record che contiene uno o più campi di tipo ↑T, allora possono essere costruite strutture equivalenti a grafi finiti arbitrari, in cui i T rappresentano i nodi ed i puntatori gli archi.

Come esempio si consideri la costruzione di una "banca dati" per un certo gruppo di persone. Si supponga che le persone siano rappresentate dai records

visti nel settimo capitolo. Si può quindi formare una catena o lista di tali records aggingendo un campo di tipo puntatore come è mostrato qui di seguito:

```
type lega = ↑persona;
    ...
    persona = record
                ...
                prossimo : lega;
                ...
            end;
```

Una lista di n persone può essere rappresentata come in figura 10.a.

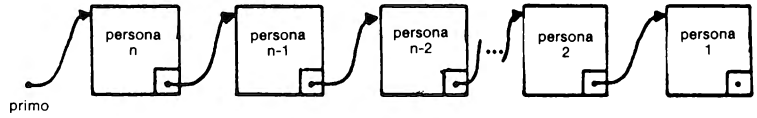


Fig. 10.a - Lista concatenata.

Una varibile di tipo lega, chiamata "primo" punta al primo elemento della lista. Il puntatore dell'ultima persona è **nil**.

Se supponiamo che il flusso "input" contiene n codici fiscali, il codice che segue potrebbe essere stato usato per cotruire la catena:

```
var   prima, p : lega; i : integer
            ...
      prima := nil;
      for   i := 1 to n do
      begin   read(s); new(p);
          p↑. prossimo := prima;
          p↑. cod := s;
          prima := p;
      end
```

Per accedere ai records introduciamo una nuova variabile, che chiamiamo pt, di tipo lega che permetta di muoversi liberamente attraverso la lista. Supponiamo poi di voler accedere ad una persona con codice xy10. Il metodo è di far avanzare pt lungo la lista fino a che si è individuata la persona desiderata:

```
pt := prima;
while   pt↑. cod <> xx10 do pt := pt↑. prossimo
```

Detto a parole sarebbe: "Si faccia puntare pt al primo elemento. Finché il codice fiscale dell'elemento puntato da pt non è uguale a xy10, si faccia avanzare pt sulla variabile indicata dal puntatore contenuto nel campo "prossimo" del record puntato in quel momento da pt." Si noti che

```
primo↑. prossimo↑. prossimo
```

acceda alla terza persona.

Naturalmente questa semplice istruzione di ricerca è corretta solo se si è sicuri che c'é almeno una persona con codice fiscale xy10 nella lista. È questa un'assunzione realistica? È sempre necessaria una verifica per assicurarsi di riconoscere la fine della lista. Si potrebbe tentare la seguente soluzione:

```
pt := prossimo;
while (pt <> nil) and (pt↑. cod <> xy10) do pt := pt↑. prossimo
```

Ma ricordando anche il quarto capitolo, si nota che se pt = **nil**, la variabile pt ↑, a cui di fa riferimento nel secondo fattore della condizione di terminazione non esiste. Riportiamo due possibili soluzioni che trattano correttamente questa situazione:

```
(1)   pt := prima; b := true;
      while (pt <> nil) and b do
          if pt↑. cod = xy10 then b := false else pt := pt↑. prossimo

(2)   pt := prima;
      while pt <> nil do
      begin if pt↑. cod = xy10 then goto 13;
          pt := pt↑. prossimo
      end
```

Consideriamo ora un altro problema: aggiungere alla lista una nuova persona. Va dapprima, allocato spazio in memoria e creato un riferimento con la procedura standard *new*.

new (p)	alloca una nuova variabile v e ne assegna il puntatore alla variabile p. Se v è di tipo record con variante. La procedura new (p) alloca memoria sufficiente per contenere tutte le varianti. La forma
new (p, t1,...,tn)	può essere usata per allocare una variabile di grandezza appropriata per la variante con valori del campo selezionante uguale alle costanti t1,...,tn. I valori del

campo selezionante devono essere elencati in modo contiguo e nell'ordine con cui sono dichiarati. Tutti i campi selezionati "trailing" possono essere omessi. Questo *non* implica assegnamenti ai campi selezionanti.

Attenzione: se una variabile record p è creata con la seconda forma della procedura new, questa variabile non deve cambiare le sue varianti durante l'esecuzione del programma. L'assegnamento all'intera variabile non è permesso; tuttavia si possono fare assegnamenti alle componenti di p ↑.

Il primo passo nel programmare una soluzione per il problema che abbiamo posto sopra, è di introdurre una variabile puntatore che chiameremo newp. L'istruzione

```
new (newp)
```

allorché una nuova variabile di tipo persona.

Come prossimo passo si deve inserire la nuova variabile puntata da newp, dopo l'elemento a cui fa riferimento pt. Si veda la figura 10.b.

L'inserzione è, semplicemente, un cambio di puntatori:

```
newp↑. prossimo := pt↑. prossimo;
pt↑. prossimo := newp
```

La figura 10.c mostra il risultato.

La cancellazione di un elemento che segue il puntatore ausiliario pt è realizzata con un'unica istruzione:

```
pt↑. prossimo := pt↑. prossimo ↑. prossimo
```

È spesso pratico quando si lavora su una lista usare due puntatori, con uno che segue l'altro. Nel caso di una cancellazione è quindi probabile che un puntatore - supponiamo p1 - precede l'elemento da cancellare e p2 punta all'elemento stesso. La cancellazione di un elemento può essere espressa con:

```
p1↑. prossimo := p2↑. prossimo;
```

Bisogna tuttavia fare attenzione che in parecchi sistemi una cancellazione fatta in questo modo porta ad una perdita di memoria usabile (libera). Un'alternativa possibile è quella di mantenere una lista di elementi "cancellati". Le nuove

variabili saranno quindi prese da questa lista invece di chiamare la procedura new. La cancellazione di un elemento diventa ora un trasferimento di questo

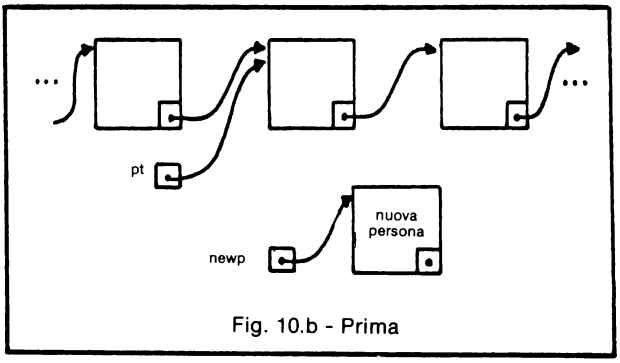


Fig. 10.b - Prima

elemento sulla lista degli elementi liberi.

```
p1↑. prossimo := p2↑. prossimo;
p2↑. prossimo := libero;
libero := p2
```

Le *liste concatenate* sono la rappresentazione più efficiente per inserire e cancellare elementi. I vettori richiedono lo slittamento giù (su) di qualsiasi

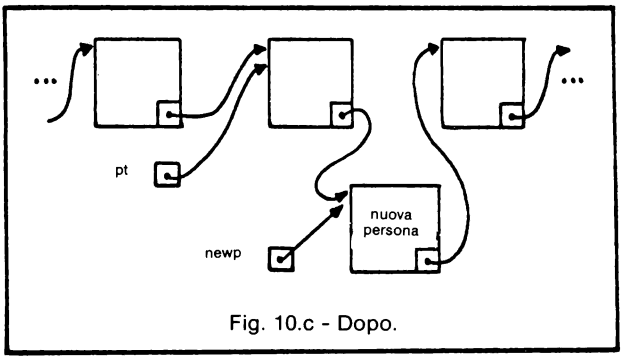


Fig. 10.c - Dopo.

elemento sotto l'indice nel caso di inserimento (cancellazione), ed i flussi richiedono una riscrittura completa.
Si faccia riferimento al capitolo 11 (programma 11.5) per un esempio che tratta una struttura ad albero invece di una lista lineare.

Un consiglio

Pascal fornisce una grande varietà di strutture dati. Tocca al programmatore analizzare il suo problema in modo tale da determinare la struttura dati più abatta sia per rappresentare la situazione che per valutare l'algoritmo. Come è stato indicato nell'esempio della "banca dati", le liste concatenate sono, soprattutto, indicate per inserimenti e cancellazioni. Se queste operazioni avvengono poco frequentemente, ed è molto importante un accesso efficiente, allora è più appropriata la rapresentazione con un vettore di records.

CAPITOLO 11

PROCEDURE E FUNZIONI

I primi miglioramenti nella tecnica della programmazioe portano a costruire programmi per *raffinamenti successivi*. Ad ogni passo il programmatore spezza il suo lavoro in un certo numero di "funzioni", definendo in questo modo dei programmi parziali. Anche se è possibile, è preferibile non mascherare questa struttura. Il concetto di *procedura* (o *sottoprogramma chiuso*) permette di evidenziare gli "spezzoni di lavoro" come sottoprogrammi.

A. Procedure

La *dichiarazione di procedura* serve per definire una parte di programma ed associarla con un identificatore, di modo che possa essere attivata con *chiamata* (istruzione) di procedura. Una dichiarazione si differenzia, come forma, da un programma solo per *l'intestazione di procedura*. Si ripensi alla parte di programma che ricercava i valori minimi e massimi in un elenco di interi. Riportiamo ora un programma che, come estensione del precedente, somma ad a [1] ... a [n] incrementi di j1 ... jn e che usa una procedura per calcolare il minimo ed il massimo.

```
{ programma 11.1
  estensione del programma 6.1 }
program minmass2 (input, output);
const n = 20;
var a : array [I..n] of integer;
    i, j : integer;
procedure minmass;
    var i : 1..n; u, v, min, mass : integer;
begin min := a [1]; mass := min; i := 2;
    while i < n do
    begin u := a [i]; v := a [i+1];
        if u > v then
        begin if u > mass then mass := u;
                if v < min then min := v
        end else
        begin if v> max then max := v;
                if u < min then min := u
        end;
        i := i+2
    end;
    if i = n then
        if a[n] >mass then mass : = a [n]
        else if a [n] < min then min := a [n];
    writeln (min, mass); writeln
end; {min, mass}

begin {legge vettore}
    for i := 1 to n do
        begin read (a [i]: write (a [i] :3)
        end;
    writeln;
    minmass;
    for i := 1 to n do
        begin read (j); a [i] := a [i] +j; writeln (a [i]: 3)
        end;
    writeln;
    minmass
end.
```

```
-1  -3 4   7 8 54 23-5 3   9   9  9-6 45 79 79 3 1 1   5
      -6          79

44 40 7 15 9 88 15-4 7 43 12 17-7 48 59 39 9 7 7 12
      -7          88
```

Benché semplice, questo programma permette di chiarire parecchi punti:

1. La forma più semplice di INTESTAZIONE di PROCEDURA, cioè: procedura <identificatore>;

2. VARIABILI LOCALI. Le variabili i, u, v, min e mass sono locali alla procedura minmass. Si può quindi riferirsi loro solo all'interno dell'ambiente di minmass; assegnamenti a queste variabili non hanno alcun effetto sul programma al di fuori dell'ambiente di minmass.

3. VARIABILI GLOBALI. a, i e j sono variabili globali. A queste variabili si può fare riferimento in ogni parte del programma. (p.e. il primo assegnamento in minmass è min := a [1])

4. PRECEDENZA DEI NOMI. Si noti che i è il nome sia di una variabile locale che di una globale. Naturalmente non sono la stessa variabile! Una procedura può accedere a qualsiasi variabile globale, o può ridefinire il nome. Se il nome di una variabile è ridefinito, la nuova associazione nome/tipo è valida nell'ambiente della procedura in cui è fatta la definizione, e la variabile globale con quel nome (a meno che sia passato come parametro) non è più disponibile nella procedura. Gli assegnamenti all'i locale (p. e. i := i+2) non ha effetto sulla i globale; e poiché i denota la variabile locale, la variabile globale i è effettivamente inaccessibile.

   È un buon metodo di programmazione quello di dichiarare tutti gli identificatori a cui non si fa riferimento fuori di una procedura, come strettamente locali alla stessa. Non solo ciò permette una buona documentazione ma permette una maggior sicurezza. La variabile i potrebbe, ad esempio, essere stata lasciata globale; ma un ulteriore estensione del programma che chiamasse la procedura minmass all'interno di un anello controllato da i portebbe ad un'elaborazione non corretta.

5. ISTRUZIONI PROCEDURA. In questo esempio l'istruzione "minmass", nel programma principale, attiva la procedura.

Esaminando ulteriormente l'ultimo esempio si nota che minmass è chiamata due volte. Formulando una parte del programma come procedura (non scrivendo cioé esplicitamente due volte quella parte) il programmatore non solo risparmia tempo di battitura ma anche spazio in memoria. Il codice statico è memorizzato una unica volta, e lo spazio che definisce le variabili è attivato solo durante l'esecuzione della procedura.

Non bisognerebbe, tuttavia, esitare nel formulare un'azione come procedura - anche se chiamate una sola volta - se ciò aumenta la leggibilità. Il definire i successivi passi di raffinamento come procedure rende il programma più comunicabile e verificabile.

Quando si scompone un programma in sottoprogrammi è spesso necessario introdurre nuove variabili per rappresentare gli argomenti ed i risultati dei sottoprogrammi. Lo scopo di queste variabili dovrebbe essere chiaro dal testo del programma.

Il programma che segue estende l'esempio precedente per calcolare i valori minimi e massimi di un vettore in un modo più generale.

```
{ programma 11,2
  estensione del programma 11.1 }

program minmass 3 (input, output);
const n = 20;
type lista = array [1..n] of integer;
var a, b : lista;
    i, min1, min2, mass1, mass2, : integer;
procedure minmass (var g:lista; var j, k:integer);
    var i:1..n; u, v,:integer;
begin j := g [1]; k := j; i := 2;
    while i < n do
    begin u := g [i]; v := g [i+1];
        if u > v then
        begin if u > k then k := u;
              if v < j then j := v
        end else
        begin if v > k then k := v;
              if u < j then j := u
        end;
        i := i+2
    end;
    if i=n then
        if g [n] > k then k := g [n]
        else if g [n] < j then j := g [n];
end; {minmass}
```

```
begin {lettura vettore}
    for i := 1 to n do
        begin read (a[i]); write (a[i]:3) end;
    writeln;
    minmass (a, min, 1 mass1);
    writeln (min1, mass1, mass1-min1); writeln;
    for i := 1 to n do
        begin read (b[i]); write (b[i]:3) end;
    writeln;
    minmass (b, min2, mass2);
    writeln (min2, mass2, mass2-min2);
    writeln (abs(min1-min2), abs (mass1-mass2)); writeln;
    for i := 1 to n do
        begin a[i] := a[i]+ b[i]; write (a[i]:3) end;
    writeln;
    minmass (a, min1, mass1);
    writeln (min1, mass1, mass1-min1)
end
```

```
-1 -3 4  7 8 54 23 -5 3  9  9  9 -6 45 79 79 3 1 1  5
      -6        79      85

45 43 3  8 1 34 -8 1 4 34  3  8 -1  3 -2 -4 6 6 6  7
      -8       45      53
       2       34

44 40 7 15 9 88 15 -4 7 43 12 17 -7 48 77 75 9 7 7 12
      -7        88      95
```

La procedura del programma 11.2 ha un'itestazione della seconda forma:

```
procedure <identificatore>  ( <sezione parametri formali>
                            {; <sezione parametri formali> } );
```

La sezione dei parametri formali elenca il nome di ogni parametro formale seguito dal suo tipo ed è seguita dalla parte di dichiarazione che introduce gli oggetti locali alla procedura.

Le etichette nella parte di definzione delle etichette, gli identificatori introdotti nella sezione dei parametri formali, la parte definizione costanti, la parte definizione tipi, le parti di dichiarazione delle variabili, procedure o funzioni sono *locali* alla dichiarazione di procedura che è l'*ambiente* di questi oggetti. Tutti questi elementi non sono conosciuti al di fuori del loro ambiente. Nel caso di variabili locali, il loro valore è indefinito all'inizio della parte istruzioni.

I *parametri* forniscono un meccanismo di sostituzione che permette la ripetizione di un processo con argomenti cambiati. (La procedura minmass è, ad esempio, stata chiamata due volte per esaminare il vettore a ed una per il b).

Esiste una corrispondenza tra l'intestazione di una procedura e l'istruzione procedura. La seconda contiene un elenco di *parametri effettivi*, che sostituiscono i corrispondenti *parametri formali* che sono definiti nella dichiarazione della procedura. La corrispondenza tra parametri formali ed effettivi è di tipo posizionale. Ci sono quattro generi di parametri indicati come: parametri valore, variabile, procedura (il parametro effettivo è un identificatore di procedura) e funzione (il parametro effettivo è un identificatore di funzione).

Il programma 11.2 mostra il caso di un *parametro variabile*. Il parametro effettivo *deve essere una variabile*; il parametro formale corrispondente deve essere preceduto dal simbolo *var* e rappresenta la variabile effettiva durante tutta l'esecuzione della procedura. Inoltre, se x1...xn sono le variabili effettive che corrispondono ai parametri formali v1...vn, allora x1...xn devono essere variabili *distinte*.

Tutti gli indirizzi sono calcolati al momento della chiamata della procedura. Perciò, se una variabile è un elemento di un vettore, l'espressione del suo indice è valutata quando la proceura è chiamata.

Per rappresentare con un'immagine l'allocazione della memoria, si potrebbe disegnare una freccia per ogni variabile che va dal nome del parametro formale alla locazione di memoria del corrispondente parametro effettivo. Ogni operazione che tratta un parametro formale è quindi fatta direttamente sul parametro effettivo. Tutte le volte che un parametro rappresenta il *risultato* di una procedura - come per j e k dell'esempio precedente - deve essere definito come un parametro variabile.

Quando, nell'intestazione, i parametri non sono preceduti da alcun simbolo sono detti *parametri valore*. In questo caso il parametro effettivo *deve essere un'espressione* (di cui una variabile è un caso particolare). Il corrispondente parametro formale rappresenta una variabile nella procedura chiamata. Essa assume, come valore iniziale, il valore corrente del corrsipondente parametro effettivo: il valore, cioé, dell'espressione quando è chiamata la procedura. La procedura, con degli assegnamenti, può quindi cambiare i valori di questa variabile; questo non può, tuttavia, influenzare il valore del parametro effettivo. Perciò un parametro valore *non può mai* rappresentare i risultato di un calcolo.

Il programma 11.3 mostra il diverso effetto degli assegnamenti su parametri valore e variabile.

```
{ programma 11.3
  parametri delle procedure }


program    parametri (output);

var    a,b : integer
procedure    h (x : integer; var y : integer);
begin    x := x+1; y := y+1;
     writeln (x,y)
end;
begin    a := 0; b := 0;
     h (a,b)
     writeln (a,b)
end.

          1          1
          0          1
```

Nel programma 11.2 non è cambiato nessuno dei valori del vettore g; g non è un risultato. Il risultato finale sarebbe perciò rimasto immutato se g fosse stato definito come un parametro valore. È molto utile pensare all'implementazione per capire perché non è stato scelto questo tipo di parametro.

Una chiamata di una procedura alloca una *nuova* area per ogni parametro valore; questa rappresenta la variabile locale. Il valore corrente del parametro effettivo è "copiato" in questa locazione; l'uscita della procedura causa, semplicemente, il rilascio dell'area.

Se un parametro non è usato per trasferire il risultato di una procedura, è generalmente preferito un parametro valore. Si può far riferimento più velocemente e si è protetti da erronee modifiche dei dati. tuttavia nei casi in cui il parametro è di tipo strutturato (p.e. un vettore), bisogna fare attenzione in quanto l'operazione di ricopiatura è relativamente pesante e la quantità di memoria necessaria per allocare i dati può essere grande. Poiché il riferimento ad ogni elemento del vettore è fatto una sola volta, è preferibile definire il parametro come parametro variabile.

Si possono cambiare le dimensioni di un vettore con una semplice redifinizione di n. Per rendere il programma adatto ad un vettore di numeri reali, basta cambiare il tipo e la definizione delle variabili; le istruzioni non sono influenzate dal fatto che i dati sono degli interi.

L'uso dell'identificatore di procedura nel testo della procedura stesa ne implica l'esecuzione *recursiva*. Problemi che si prestano a definizioni recursive, hanno spesso soluzioni di tipo recursivo. Il programma 11.4 ne è un esempio.
In accordo con le sintassi rappresentata in fig. 11.a sono date le espressioni seguenti (un punto indica la fine dell'ultima espressione):

(a+b)*(c-d)
a+b*c-d
(a+b)*(c-d)
a+b*c-d
a*a*a*a
b+c*(d+c*a*a)*b+a.

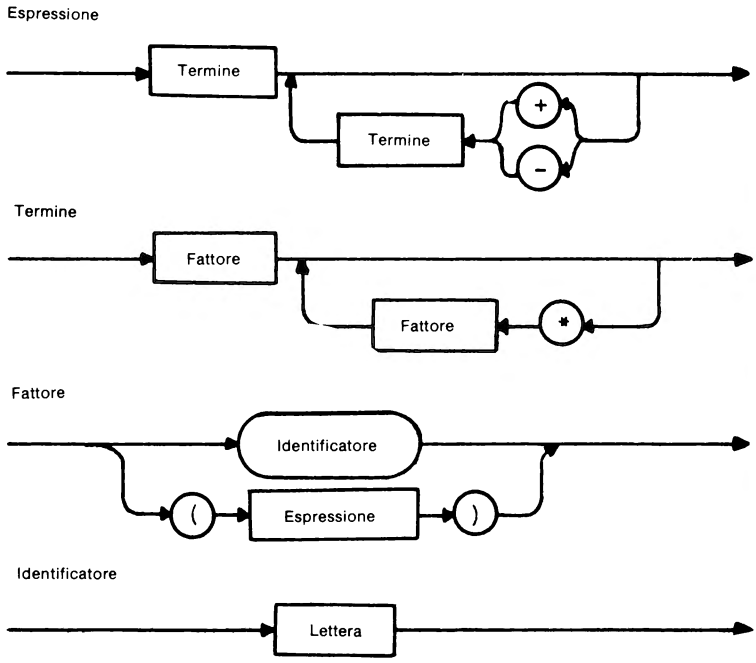


Fig. 11.a - Espressioni.

Si tratta di costruire un programma che converte le espresioni nella forma postfix (notazione Polacca). Questo è realizzato costruendo una procedura di conversione per ognuno dei costrutti sintattici: espressione, termine, fattore. Poiché questi costrutti sintattici sono definiti recursivamente, le procedure corrispondenti possono chiamarsi recursivamente.

```
{ programma 11.4
  conversione nella forma postfix }
program   postfix (input, output);

var   ch : char;

procedure   trova;
begin repeat   read (ch)
     until   (ch <> ' ') and not eoln (input)
end;
procedure   espressione;
     var   op : char;
     procedure   term;
     procedure   fattore;
     begin if   ch = '(' then
              begin   trova; espressione; { ch = ) }
              end else   write (ch);
              trova
     end;   {fattore}
     begin   fattore;
              while   ch = '*' do
              begin   trova; fattore; write (' * ')
              end
     end;   {term}

begin   term;
     while   (ch = '+') or (ch = '-') do
          begin   op := ch; trova; term; write (op)
          end
end;   {espressione}
begin   trova;
     repeat   write   (' ');
          espressione;
          writeln
     until   ch = '.'
end.
ab+cd-*
abc*+d-
ab+c*d-
abcd-*+
aa*a*a*
bcdca*a*+*b*+a+
```

Un *albero binario* è una struttura dati che è naturalmente definita recursivamente ed elaborata con algoritmi recursivi. È costituito da un insieme finito di nodi ed è vuoto o costituito da un nodo (radice) con due alberi binari disgiunti, chiamati sottoalbero sinistro e destro [6]. Procedure recursive per generare ed attraversare alberi binari riflettono questo modo di definirili.

Il programma 11.5 costruisce un albero binario e lo attraversa in pre-,in-, e postordine. L'albero è specificato in preordine: elencando cioé i nodi (in questo caso con una lettera) partendo dalla radice e seguendo prima il sottoalbero di sinistra e quindi quello di destra così che l'input che rappresenta la fig. 11.6 è:

abc..de..fg..hi..jkl..m..n..

dove un punto indica un sottoalbero vuoto.

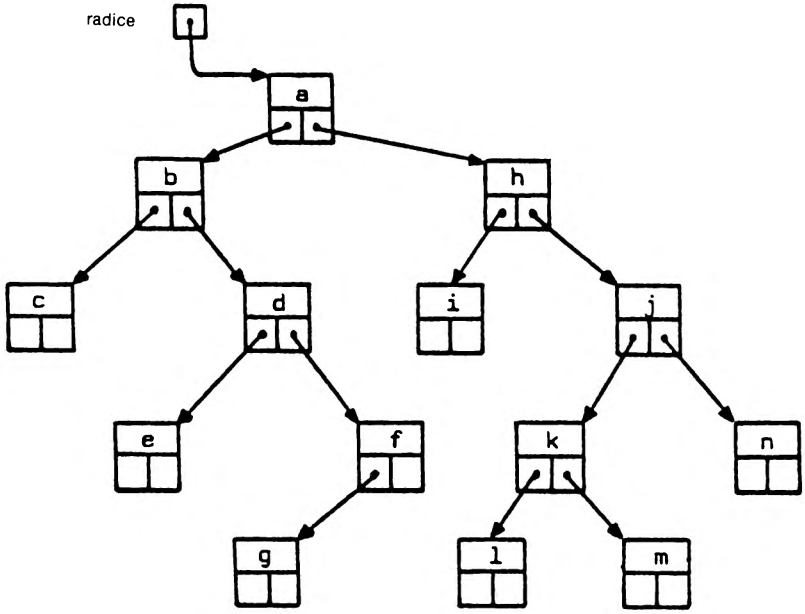


Fig. 11.b - Struttura albero binario.

```
{ programma 11.5
  attraversamento di un albero binario }

program    attraversamento (input, output);

type    ptr    = ↑nodo;
         nodo = record    info : char;
                       sleg, dleg : ptr
                    end;
var    rad : ptr; ch : char;

procedure    preordine (p:ptr);
begin   if  p < > nil then
        begin    write (p ↑ .info);
                 preordine (p ↑ .sleg);
                 preordine (p ↑ .dleg)
        end
end;    {preordine}

procedure    inordine (p:ptr);
begin   if  p <> nil then
        begin    inordine (p ↑ .sleg);
                 write (p ↑ .info);
                 inordine (p ↑. dleg)
        end
end;    {inordine}

procedure    postordine (p : ptr);
begin   if  p <> nil then
        begin    postordine (p ↑ .sleg);
                 postordine (p ↑. dleg.);
                 write (p ↑. info)
        end
end;    {postordine}

procedure    enter (var p : ptr);
begin    read (ch); write (ch);
     if    ch < > '.' then
     begin    new (p);
          p ↑. info := ch;
          enter (p ↑. sleg);
          enter (p ↑. dleg);
     end
     else    p := nil
end;    {enter}
```

```
begin
    write (' '); enter (rad); writeln
    write (' '); preordine (rad); writeln;
    write (' '); inordine (rad); writeln;
    write (' '); postordine (rad); writeln
end.
```

```
abc..de..fg...hi..jkl..m..n..
abcdefghijklmn
cbedgfaihlkmjn
cegfdbilmknjha
```

Sebbene le tecniche recursive appaiono "intelligenti", non sempre portano a soluzioni efficienti per cui il lettore è messo in guardia contro un uso indiscriminato delle stesse.

Se una procedura P chiama una procedura Q che a sua volta chiama P, allora sia P che Q devono essere "pre-segnalate" con una *dichiarazione in avanti.* (sezione 11.C)

Le *procedure standard* riportate nell'appendice A sono definite in ogni implementazione di Pascal. Le varie implementazioni di Pascal possono aggiungere delle procedure predefinite e poiché devono essere, come tutti gli oggetti standard, dichiarate in un ambiente che circonda il programma utente, non sorgono conflitti se, all'interno del programma, una dichiarazione ridefinisce gli stessi identificatori. Le procedure standard get, put, read, write, reset e rewrite sono state introdotte nel nono capitolo. Nel capitolo 12 si ridiscuteranno read e write.

B. Funzioni

Le *funzioni* sono sottoprogrammi (nello stesso senso delle procedure) che calcolano un singolo valore - scalare o puntatore - da usarsi nel computo di un'espressione.
Un *indicatore di funzione* specifica l'attivazione di una funzione ed è costituito dall'identificatore di funzione e dall'elenco dei parametri effettivi. I parametri sono variabili, espressioni, procedure o funzioni e sostituiscono i corrispondenti parametri formali.

La dichiarazione di una funzione ha la stessa forma di un programma, eccettuata l'*intestazione di funzione* che ha la forma:

```
function <identificatore> : <tipo del risultato>;
```

oppure

```
function <identificatore> ( sezioni parametri formali>
         {; <sezione parametri formali> } ) : <tipo del risultato>;
```

Come per le procedure, le etichette nella parte di definizione etichette e tutti gli identificatori introdotti nella parte dei parametri formali, la parte definizione costanti, la parte definizione tipi, le parti di dichiarazioni di variabili, procedure o funzioni sono *locali* alla dichiarazione della funzione, che è chiamata l'*ambiente* di questi oggetti. Tutti questi oggetti non sono conosciuti al di fuori del loro ambiente. I valori delle variabili locali sono indefiniti all'inizio della parte istruzioni.

L'identificatore specificato nell'intestazione della funzione dà il nome alla funzione. Il tipo del risultato può essere scalare, sottocampo o puntatore. All'interno della dichiarazione della funzione ci deve essere un assegnamento all'identificatore di funzione: è con questo assegnamento che si "ritorna" il risultato della funzione.

Negli esempi riportati sino a questo punto si sono visti solo parametri valori e variabili, sono comunque possibili anche parametri funzioni e procedure. Entrambi devono essere individuati da un simbolo speciale; il simbolo **procedure** caratterizza un parametro formale funzione. Il programma che segue trova lo zero di una funzione con il metodo della bisezione; la funzione è specificata al momento della chiamata.

```
{ programma 11.6
  trova lo zero di una funzione con bisezione. }

program    bisez (input, output);

const    eps = 1e - 14;
var    x,y : real;

function    zero (function f : real; a,b : real) : real;
    var    x,z : real; s : boolean;
begin    s := f(a)<0;
    repeat    x := (a+b) / 2.0;
        z := f(x);
        if    (z<o) = s then a := x else b := x
    until    abs (a-b)<eps;
    zero := x
end;    {zero}
begin    {main}
    read (x,y); writeln (x,y, zero (sin,x,y));
    read (x,y); writlen (x,y,zero(cos,x,y))
end.
```

```
—1.000000000000e+00    1.000000000000e+00    —7.105427357601e—15
  1.000000000000e+00    2.000000000000e+00      1.570796326795e +00
```

Un assegnamento (all'interno di una dichiarazione di funzione) ad una variabile non-locale o ad un parametro variabile è detto *effetto margine*. Questi assegnamenti mascherano spesso lo scopo di un programma e complicano terribilmente la verifica. Non bisognerebbe perciò *mai* usare funzioni con effetto margine (collaterale).

Si consideri come esempio il programma 11.7

```
{ programm 11.7
  test effetto margine }

program    effettomarg (output);

var    a,z : integer;

function    strisciante (x : integer) : integer;
begin    z := z-x; {effetto margine su z}
    strisciante := sqr (x)
end;
begin
    z := 10; a := strisciante (z); writeln (a,z);
    z := 10; a := strisciante (10) * strisciante (z); writeln (a,z);
    z := 10; a := strisciante (z) * strisciante (10); writeln (a,z);
end.
```

```
      100          0
        0          0
    10000        —10
```

Il prossimo esempio formula l'algoritmo esponenziale del programma 4.8 come una dichiarazione di funzione.

```
{ programma 11.8
  estensione programma 4.8 }

program    espon 2 (output);

var    pi, spi : real;

function    potenza (x:real; y:integer) : real; {y >= 0}
    var    z : real;
begin    z := 1;
    while    y >0 do
    begin
        while not    odd (y) do
        begin    y := y div 2; x := sqr (x)
        end;
    y := y-1; z := x*z
    end;
    potenza := z
end; {potenza}

begin    pi := 3.14159;
    writeln (2, 0, 7, potenza (2, 0, 7));
    spi := potenza (pi, 2);
    writeln (pi, 2, spi);
    writeln (spi, 2, potenza (spi, 2));
    writeln (pi, 4, potenza (pi, 4))
end.
```

```
2.000000000000e+00        7        1.280000000000+02
3.141590000000e+00        2        9.869587728100e+00
9.869587728100e+00        2        9.740876192266e+01
3.141590000000e+00        4        9.740876192266e+01
```

Il fatto che l'identificatore di funzione compaia in un'espressione all'interno della funzione implica un'esecuzione *recursiva* della stessa.

```
{ programma 11.9
  formulazione recursiva di gcd }


program    gcdrecursivo (output);


var    x,y,n : integer;
function    gcd (m,n : integer) : integer;
begin if    n=0 then gcd := m
      else    gcd := gcd (n,m mod n)
end;    {gcdrecursivo}

procedure    prova (a,b : integer);
begin    writeln (a,b,gcd(a,b))
end;

begin    prova (18,27);
         prova (312,2142);
         prova (61,53);
         prova (98,868)
end.
```

18	27	9
312	2142	6
61	53	1
98	868	14

La chiamata di una funzione può avvenire prima della sua definizione se c'é un *riferimento in avanti* (sezione 11.C).

Le *funzioni standard* riportate in appendice A sono definite in ogni implementazione di Pascal. Le varie implementazioni di Pascal possono aggiungere delle procedure predefinite.

C. Note

1. Le chiamate di procedure (funzioni) possono essere fatte prima della definizione della procedura (funzioni) se c'é un *riferimento in avanti*. Ecco un esempio:

```
procedure    Q (x:T); forward;
procedure    P (y:T);
     begin
          Q (a)
     end;
procedure    Q: {i parametri non sono ripetuti}
     begin
          P (b)

     end;
begin
          P (a);
          Q (b);
end.
```

Si noti che l'elenco dei parametri e il tipo del risultato finale sono scritti *solo* nel riferimento in avanti.

2. Le procedure e le funzioni che sono usate come parametri per altre procedure e funzioni devono verificare durante l'esecuzione se il parametro è chiamato per valore o indirizzo.

3. Un componente di una struttura impaccata non può apparire come un parametro effettivo variabile. Non è quindi necessario passare indirizzi di parti di parole e verificare durante l'esecuzione la rappresentazione interna della variabile effettiva.

4. Parametri flusso devono essere specificati come **var**-parametri.

CAPITOLO 12

INGRESSO E USCITA

Al problema della comunicazione tra uomo ed elaboratore si è già fatto cenno nel capitolo 9. Entrambi imparano a comprendere attraverso ciò che è chiamato *riconoscimento di modelli.*
Sfortunatamente i modelli riconosciuti piu facilmente dall'uomo (soprattutto disegni e suoni) sono molto diversi da quelli accettabili da un elaboratore (impulsi elettrici). In effetti il costo della trasmissione dei dati--che implica la traduzione di modelli leggibili per l'uomo in altri comprensibili per l'elaboratore e viceversa-- può essere alto quanto l'elaborazione delle informazioni trasmesse. Per questo molte risorse sono dedicate alla ricerca di metodi di traduzione automatica per minimizzare i costi. Il lavoro di comunicazione è chiamato gestione dell'ingresso ed uscita: I/0.

L'uomo può inviare le informazioni per mezzo di dispositivi di ingresso (p.e. lettori di schede, bande perforate, nastri magnetici, terminali) e riceverle tramite dispositivi di uscita (p.e. stampanti, perforati di schede e bande di carta, terminali, unità video). Comuni ed antrambi - e caratteristici di ogni installazione - è un insieme di caratteri leggibili (capitolo 2). Pascal definisce i due flussi standard *input* e *output*, usati come parametri del programma, su questo insieme di caratteri (si veda anche il capitolo 9). Si può accedere ai flussi-testo con le procedure standard get e put. Questo è naturalmente molto scomodo dal momento che queste procedure sono state definite per gestire un solo carattere. Come caso pradigmatico si può supporre di avere un numero naturale memoriazzato in una variabile x e di volerlo stampare nel flusso output. Si noti che il tipo di carattere che denota la rappresentazione decimale di un valore sarà molto diversa da quella denotante un valore scritto in numeri romani (si veda il programma 4.7). Ma poiché si è solitamente interessati ad una rappresentazione decimale, sembra sensato dare delle procedure interne standard che trasformano numeri astratti (qualunque sia la rappresentazione interne usata) in una sequenza di cifre decimali e viceversa.

Le due procedure standard *read* e *write* offrono la possibilità di analizzare e di costruire flussi-testo. Le sintassi per chiamare queste procedure non è standard: possono essere usate con un numero variabile di parametri i cui tipi non sono fissi.

A. Procedura read

Si supponga che v1, v2, ... ,vn denotino variabili di tipo carattere, intero o reale, e f denoti un flusso-testo.

1. read (v1, ... ,vn) sta per
   read (input, v1, ... ,vn)
2. read (f, v1, ... ,vn) sta per
   begin read (f, v1); ... ;read (f, vn) **end**
3. readln (v1, ... ,vn) sta per
   readln (input, v1, ... ,vn)
4. readln (f, v1, ... vn); sta per
   begin read (f, v1); ...; read (f, vn); readln (f) **end**

   L'effetto è che, dopo che è stato letto vn (dal flusso-testo f), è tralasciato il resto della linea corrente.
   (Tuttavia, i valori di v1 ... vn possono estendersi su parecchie linee)
5. Se ch è una variabile di tipo carattere, read (f, ch) sta per:

   begin ch := f↑; get (f) **end**

   Se un parametro v è di tipo intero (o sottocampo), o reale viene letta una sequenza di caratteri che rappresenta un numero reale o intero in accordo con la sintassi di Pascal. (Numeri consecutivi devono essere separati da spazi o fine linea)

esempi:

Si leggano ed elaborino una sequenza di numeri il cui ultimo valore è immediatamente seguito da un asterisco. Si supponga che f sia un flusso-testo, x e ch siano variabili di tipo, rispettivamente, intero (o reale) e carattere.

```
reset (f)
repeat   read (f, x, ch);
         P (x)
until    ch = '*'
```

Una situazione certamente più comune è quella in cui non si conosce il numero dei dati da leggere, e non c'é un simbolo speciale che indica la fine dell'elenco. Si riportano ora due possibili schemi; nel primo sono elaborati

dati singoli, nel secondo n-ple di numeri.

```
reset (f);
while not    eof (f) do
begin    read (f, x); saltaspazi (f);
         P (x)
end
```

dove saltaspazi (f) sta per:

```
while    (f↑ <> ' ') and not eof (f) do get (f)
```

Il secondo schema elabora n-pie di numeri:

```
reset (f)
while not    eof (f) do
begin    read (f, x1, ... , xn); saltaspazi (f);
         P (x1, ... ,xn)
end
```

(Affinché il secondo schema funzioni correttamente, il numero totale di dati deve essere multiplo di n).
La procedura read può anche essere usata per leggere da un flusso f che non è un flusso-testo.

```
read (f, x)
```

in questo caso sta per:

```
begin    x := f↑; get (f) end
```

B. Procedura write

La procedura write aggiunge stringhe di caratteri (uno o più caratteri) ad un flusso-testo. Siano p1, p2, ... ,pn dei parametri della forma definita sotto (si veda 5), ed f un flusso-testo. Allora:

1. write (p1, ... ,pn) sta per:
   write (output, p1, ..., pn)

2. **write** (f, p1, ... , pn); ... ; sta per:
   begin write (f, p1); ... ; write (f, pn) **end**

3. writeln (p1, ... , pn) sta per:
   writeln (output, p1, ... , pn)

4. writeln (f, p1, ... , pn) sta per:

   begin write (f, p1); ... ; write (f, pn); writeln (f) **end**

   Come effetto si ha la scrittura di p1, ... ,pn e la terminazione della linea corrente del flusso-testo f.

5. Ogni parametro pi deve essere di una delle forme:

   e
   e : e1
   e : e1 : e2

   dove e, e1 ed e2 sono espressioni

6. e è il *valore* che deve essere scritto e deve essere di tipo carattere, intero, reale, booleano o può essere una stringa. Nel primo caso, write (f, c) sta per:

   f↑ := c; put (f)

7. e1, che è chiamato "lunghezza minima del campo", è un controllo opzionale. Deve essere un numero naturale ed indica il numero minimo di caratteri che devono essere scritti. Il valore e è cioé scritto con e1 caratteri (con spazi in testa). Se e1 è "troppo piccolo" è allocato più spazio. (I numeri reali devono essere scritti con almeno uno spazio in testa; questa restrizione non si applica ai numeri interi). Se la lunghezza del campo non è specificata, si assume, per difetto, un valore (dipendente dall'implementazione) in accordo con il tipo dell'espressione e.

8. e2, che è chiamato *lunghezza della parte frazionale*, è un controllo opzionale ed è applicabile solo quando è di tipo reale. Deve essere un numero naturale e specifica il numero di cifre che seguono la virgola (il punto) decimale. Il numero si dice quindi scritto in notazione virgola fissa. Se non è specificata la lunghezza della parte frazionale, il valore è stampato nella forma decimale in virgola mobile.

9. Se il valore è di tipo booleano è scritto l'identificatore standard true o false. La procedura write può essere usata anche per scrivere in un flusso f che non è un flusso-testo.

   write (f, x)

   sta in questo caso per:

   begin f↑ := x; put (f) **end**

CAPITOLO 13

PASCAL 6000 - 3.4

Lo scopo di questo capitolo è di introdurre le caratteristiche peculiari della implementazione fatta sugli elaboratori Control Data 6000. Il lettore deve tener presente che il fare affidamento su una qualsiasi delle caratteristiche peculiari di PASCAL 6000-3.4 può rendere i suoi programmi non accettabili ad altre implementazioni di Pascal. Si consiglia quindi di usare solo quelle caratteristiche descritte come *Standard Pascal* nei capitoli precedenti, soprattutto quando si vogliono scrivere programmi "portabili".

Si possono raggruppare gli argomenti di questo capitolo in quattro categorie:

A) Estensioni
B) Specifiche non definite nei capitoli precedenti
C) Restrizioni
D) Tipi, funzioni e procedure addizionali predefinite.

A. Estensioni al linguaggio Pascal

Questa sezione definisce i costrutti non standard disponibili sui sistemi Pascal 6000-3.4. Sebbene influenzati dall'ambiente determinato dal sistema operativo, sono descritti e possono essere capiti in termini indipenddenti dalla macchina.

A.1 Flussi segmentati

Un flusso può essere considerato suddiviso in *segmenti*, vale a dire una sequenza di segmenti ognuno dei quali è lui stesso una sequenza. Pascal 6000-3.4 dà la possibilità di dichiarare un flusso *segmentato*, e di riconoscere i segmenti ed i loro limiti. Ogni segmento, nella terminologia CDC SCOPE, è chiamato "*record logico*".

dichiarazione:
<tipo flusso> ::= **segmented file of** <tipo>

un esempio:

type T = **segmenti file of** char;

La funzione

eos (x) dà il valore true quando il flusso x è posizionato alla fine del segmento, altrimenti false.

Sono inoltre introdotte le due seguenti procedure standard:

putseg (x) deve essere chiamata quando è stata terminata la generazione di un segemento del flusso x, e

getseg (x) è chiamata per iniziare la lettura del prossimo segmento del flusso x. Assegna alla variabile buffer x↑ il primo componente del prossimo segmento. Se non esiste un "prossimo segmento", eof (x) diviene true; se il prossimo segmento esiste ma è vuoto, eos (x) diviene true e x↑ è indefinito. Le chiamate seguenti di *get (x)* si fermeranno sul prossimo componente o, se non esiste, ridaranno eos (x) true.

Get (x) non deve essere chiamata se sono true eos (x) o eof (x):
eof (x) implica sempre eos (x).

I vantaggi di un flusso segmentato stanno nella possibilità di posizionarsi sia in lettura che in scrittura in modo relativamente veloce su qualsiasi segmento del flusso. Allo scopo di leggere e riscrivere un flusso segmentato, le procedure standard getseg e rewrite sono estese per accettare due argomenti.

getseg (x, n) inizia la lettura dell'n^{mo} segmento, contando dalla posizione *corrente* nel flusso. n<o per un conteggio indietro; n=o implica il segmento corrente. Si noti che getseg (x, 1) è equivalente a getseg (x).

rewrite (x, n) inizia la scrittura di x all'inizio dell'n^{mo} segmento contado dalla posizione *corrente*. Si noti che rewrite (x, 1), *non* è equivalente a rewrite (x). Il secondo provoca la riscrittura partendo dall'inizio del flusso.

Poiché i flussi sono organizzati per un'elaborazione sequenziale (in avanti), non ci si deve aspettare che getseg e rewrite siano efficienti per n<=o quanto per n>o.

I due schemi di programmi che seguono con le istruzioni parametriche W,R e S mostrano le operazioni di lettura e scrittura di un flusso segmentato.

Scrittura di un flusso segmentato x:

```
rewrite (x);
repeat    {genera un segmento}
    repeat    {genera un componente}
        W (x↑); put (x)
    until    p;
    putseg (x)
until    q
```

Nota: questo schema non genererà mai un segmento od un flusso vuoti.

Lettura di un flusso segmentato x:

```
reset (x);
while not    eof (x) do
begin    {elabora un segmento}
    while not    eos (x) do
    begin    {elabora un componente}
        R (x↑); get (x)
    end;
    S; getseg (x)
end
```

Il prossimo esempio fa vedere una procedura che legge un flusso-testo segmentato f e copia le prime n linee di ogni segmento nel flusso output.

```
procedure    elenca;
    var    i, s : integer;
begin    s := 0; reset (f);
    while not    eof (f) do
    begin    s := s+1; i := 0;
        writeln    ('segmento', s);
        while not    eos (f) and (i<n) do
        begin    i := i+1; {copia una linea}
            while not    eoln (f) do
            begin    write (f↑); get (f) {prossimo carattere}
            end;
            writeln; readln (f) {prossima linea}
        end;
        getseg (f)    {prossimo segmento}
    end
end.
```

Le procedure standar *read* e *write* possono essere applicate anche a flussi segmentati.

A.2 Procedure esterne

PASCAL 6000-3.4 dà la possibilità di accedere a *procedure esterne,* vale a dire procedure (funzioni) che esistono fuori dal programma utente e che sono state compilate distintamente. Questo permette ad un programmatore Pascal di accedere a programmi di libreria. Tali procedure si dichiarano facendo seguire l'intestazione con la parola "extern" o "Fortran".

B. Specifiche lasciate indefinite nei capitoli precedenti

B.1 Intestazione dei programmi e flussi esterni

Una variabile PASCAL di tipo flusso è implementata sotto il sistema operativo CDC come flusso. I flussi locali sono allocati su disco o in una estensione della memoria di lavoro (ECS). La memoria è allocata al momento della loro generazione ed è automaticamente rilasciata quando è terminato il blocco rispetto al quale sono locali.

I flussi che esistono al di fuori del programma (vale a dire prima o dopo l'esecuzione del programma) possono essere resi disponibili se sono specificati come *parametri effettivi* nella direttiva di chiamata del programma (EXECUTE) della scheda di controllo. Sono chiamati *flussi esterni* vanno a sostituire i parametri formali specificati nell'intestazione del programma. L'intestazione ha la seguente forma:

program <identificatore> (<parametro del programma>
{, <parametro del programma> });

dove un parametro del programma è:

<identificatore flusso> o <identificatore flusso> *

I parametri sono identificatori formali di flusso; devono essere dichiarati, nel programma principale, come variabili flusso nello stesso modo in cui sono dichiarate le variabili effettive locali di tipo flusso.

I flussi denotati dai parametri formali *input* e *output* hanno caratteristiche precipue: devono essere seguite le seguenti regole:

1. L'intestazione del programma *deve* contenere il parametro output.

2. Contrariamente a tutti gli altri flussi esterni, i due identificatori formali *input* e *output non devono* essere definiti in una dichiarazione in quanto è automaticamente inserita la dichiarazione:

```
var   input, output,  text;
```

3. Le procedure reset e rewrite non hanno effetto se sono applicate ai flussi effettivi INPUT e OUTPUT

esempio:

```
program   P (output, x, y);
                ...
var   x, y : text
```

Se un parametro effettivo nella direttiva EXECUTIVE della scheda di controllo è lasciato vuoto, il corrispondente parametro formale nell'intestazione del programma è considerato come l'effettivo "nome di flusso". Per esempio se si chiama un programma con l'intestazione:

```
program   P (output, f; g)
```

allora EXECUT, (,X,) è equivalente a EXECUTE, P (OUTPUT, X, G). Si raccomanda comunque di specificare sempre completamente i parametri flusso in quanto la fiducia nei valori assegnati per difetto, porta ad errori facilmente evitabili.

B.2 Rappresentazione dei flussi

È importante, quando si ha a che fare con flussi esterni, conoscere la rappresentazione dei flussi scelta dal compilatore PASCAL. Ogni componente di un flusso PASCAL-6000, eccettuati i flussi con componenti di tipo carattere (flussi-testo), occupa un numero intero di parole da 60 bits. Per i flussi-testo, PASCAL usa la rappresentazione standard imposta dalle convenzioni CDC: in ogni parola sono impiccati 10 caratteri e questo implica che le procedure put e get includono le operazioni di impaccamento e disimpaccamento. Il fine linea è rappresentato da almeno 12 zeri appoggiati a destra. I flussi originati da lettori di *schede* seguono le stesse convenzioni dei flussi-testo. Si noti che il sistema operativo elimina parecchie (ma non tutte) spaziature intercarattere quando legge delle schede: questi flussi *non* sono quindi necessariamente costituiti da "immagini di scheda" a 80 caratteri.

I flussi che non sono segmentati sono scritti come un unico "record logico" (nella teminalogia SCOPE). Quando è letto un flusso esterno non segmentato,

i segnali di fine record sono ignorati (si veda, per una eccezione, il punto 3 che segue). Nei flussi segmentati ogni segmento corrisponde ad un "livello logico": non esiste la possibilità di specificare un "livello record".

Uso dei flussi esterni

1. Se bisogna leggere (scrivere) un flusso esterno non segmentato, l'operazione va fatta precedere da:

   reset (x) (rewrite (x))

   se il flusso è segmentato da:

   reset (x) (rewrite (x)) oppure
   getseg (x, n) (rewrite (x. n))

   (Il programmatore non deve specificare queste istruzioni nel caso in cui il programma agisca su flussi denotati da parametri formali *input output* in quanto ciò è fatto automaticamente.)

2. Ogni flusso è automaticamente "aperto" con una chiamata della ruotine OPE del sistema operativo. Un *asterisco* dopo il parametro flusso, dell'intestazione del programma, indica che il flusso è aperto solo in lettura (p.e. un flusso permanente senza permesso di scrittura). L'asterisco, in sé, non costituisce una protezione riguardo alla scrittura del flusso.

   esemio:

```
program   verdato (output, dato*);
                ...
var   dato : file of real;
      r : real;
                ...
      r := dato↑; get (dato)
                ...
```

3. Se il nome effettivo INPUT sostituisce un parametro formale di programma - p.e.f - allora f è il singolo record logico corrente del flusso INPUT.

B.3 I tipi standard

L'identificatore standard *maxint* è definito come:

```
const   maxint = 281474976710655: {= 2**48-1}
```

Il lettore deve comunque tener presente che gli elaboratori CDC non danno indicazioni di traboccamento. È perciò responsabilità del programmatore verificarlo tutte le volte che potrebbe accadere.

In realtà la macchina può memorizzare interi con valore assoluto fino a 2**59; ma in questo campo sono eseguite correttamente solo le operazioni di addizione (+), sottrazione (-), valore assoluto, moltiplicazioni e divisioni per potenze di 2 (implementate come scorrimenti) e confronti. In particolare non si può stampare un valore intero i quando abs (i) > maxint. È tuttavia possibile la verifica:

if abs (i) > maxint **then** write ('troppo grande')

REALI

Il tipo **real** è definito in accordo con il formato virgola mobile del CDC 6000: la mantissa è implementata su 48 bits che corrispondono a 14 cifre decimali. Il massimo valore assoluto è 10**322.

CARATTERI

Un valore di tipo **char** è un elemento dell'insieme dei caratteri della particolare installazione. Esistono 3 versioni:

1) Insieme dei 64 caratteri CDC scientifico
2) Insieme dei 63 caratteri CDC scientifico
3) Insieme dei 64 caratteri CDC ASCII

La tavola 13.a elenca i caratteri disponibili ed indica il loro ordinamento.

NOTA: Le specifiche CDC danno un ordinamento dei caratteri ASCII che differisce dallo standard internazionale (ISO)!

Insieme dei caratteri CDC scientifico (con 64 elementi)

	0	1	2	3	4	5	6	7	8	9
0	:	A	B	C	D	E	F	G	H	I
10	J	K	L	M	N	O	P	Q	R	S
20	T	U	V	W	X	Y	Z	0	1	2
30	3	4	5	6	7	8	9	+	—	*
40	/	(	)	$	=		,	.	≡	[
50	]	%	≠	↱	∨	∧	↑	↓	<	>
60	⩽	⩾	¬	;						

Commenti:

- 0 non è usato nella versione 63 caratteri
- 51 : nella versione 63 caratteri
- 48 ' a ETH
- 53 { a ETH
- 57 } a ETH

Insiemi di caratteri ASCII con ordinamento CDC

	0	1	2	3	4	5	6	7	8	9
0	:	A	B	C	D	E	F	G	H	I
10	J	K	L	M	N	O	P	Q	R	S
20	T	U	V	W	X	Y	Z	0	1	2
30	3	4	5	6	7	8	9	+		*
40	/	(	)	$	=		,	.	#	[
50	]	%	"	-	!	&	,	?	<	>
60	@	\	∧	;						

Fig. 13.a - Insiemi di caratteri CDC

I caratteri riportati in fig. 13.b - che fanno parte dell'insieme riportato sopra -sono accettati dal compilatore Pascal 6000-3.4 come sinonimi per i simboli standard riportati nella colonna di sinistra.

Standard Pascal	CDC scientifico	ASCII
not	¬	
and	∧	&
or	∨	
<>	≠	
<=	≤	
>=	≥	

Fig. 13b - Rappresentazione alternativa dei simboli standard.

B.4 La procedura standard "write"

Quando non è specificato il parametro li lunghezza minima del campo, si considerano, per difetto, i seguenti valori:

tipo	per difetto
intero	10
reale	22 (l'esponente è sempre espresso come: E±999)
Booleano	10
carattere	1
stringa	lunghezza della stringa
alfa (si vada D.1)	10

La fine di ogni riga in un flusso-testo f deve essere semplicemente indicata con writeln (f), dove writeln (output) può essere scritto, semplicemente, come writeln. Se un flusso-testo deve essere inviato su una stampante, nessuna linea può contenere più di 136 caratteri. Il primo carattere di ogni linea è interpretato dalla stampante come carattere di controllo e non è stampato. I possibili caratteri sono:

'+'	nessuna interlinea (sovrascritturata)
spazio	spaziatura semplice
'0'	doppia spaziatura
'1'	passaggio in testa alla prossima pagina prima di stampare.

La procedura writeln (x) è usata per segnare la fine di una linea sul flusso x. Le convenzioni del sistema operativo CDC riguardo ai flussi-testo sono tali che questa procedura genera, in certe circostanze, alcuni spazi in più. Perciò, in lettura, un flusso-testo può contenere degli spazi a fine riga che non sono mai stati scritti. (Ci piace, ma...)

C. Restrizioni

1. La parola **segmented** è riservata

2. Il tipo base di un insieme deve essere:
   a) uno scalare (o sottocampo) con al massimo 59 elementi
      o
   b) un sottocampo con il minimo maggiore o uguale a zero ed il massimo minore o uguale a 58
      o
   c) un sottocampo di tipo carattere con l'elemento massimo inferiore o uguale al valore ch (58).

3. Le funzioni e procedure standard (interne) non sono accettare come parametro effettivo. Per esempio, se si volesse far girare il programma 11.6 in PASCAL 6000-3.4, si dovrebbero scrivere le funzioni ausiliarie che seguono:

```
                ...
function    sine (x:real) : real;
    begin    sine := sin (x) end;
function    zero (function f : real; a,b : real) : real;
    begin ... end
      ...

begin
    read (x, y); writeln (x, y, zero (sine, x, y));
end.
```

4. Non è possibile costruire un flusso di flussi. Sono invece permessi records e vettori con componenti di tipo flusso.

5. Le stringhe possono essere confrontate solo se la loro lunghezza è minore di 10 o multiplo di 10.

D. Tipi, procedure e funzioni predefiniti aggiunti

D.1 Tipi predefiniti aggiunti

Il tipo *alfa* è predefinito da:

```
type    alfa = packed array [1..10] of char;
```

(Un valore di tipo alfa è rappresentabile in una parola). Le costanti di questo tipo sono *stringhe* di 10 caratteri. Agli operandi di tipo alfa sono applicabili l'assegnamento (:=) ed il confronto = e <> verificano l'eguaglianza e <, <= >=, e > verificano l'ordine in accordo con l'insieme di caratteri costituenti.

```
{ programma 13.1
  valori alfa }

program    esalfa (output);

var    n1, n2 : alfa;
begin    write ('nomi:')
     n1 := 'Gianni'; n2 := 'Katia'
     if    n2 <n1 then writeln (n2, n1)
                    else    writeln (n1, n2)
end.

nomi: katia      Gianni
```

Nota: Non è possibile leggere direttamente dei valori di tipo alfa; si suggerisce:

```
var    buf : array [1..10] of char;
       a : alfa; i : integer;
       ...
       for    e := 1 to 10 do read (buf [i]);
       pack    (buf, 1, a) {per finire read (a)}
       ...
```

D.2 Funzioni e procedure predefinite aggiunte

Procedure

date (a) assegna la data corrente alla variabile a.

halt — termina l'esecuzione del programma e scarica l'immagine post-mortem.

linelimit (f, x) — f è un flusso-testo e x è un'espressione intera. Questa procedura fa sì che il programma sia terminato se è richiesta la scrittura, nel flusso f, di un numero di linee maggiore di x.

message (x) — la stringa x è scritta nel flusso giornaliero. (x deve perciò contenere una massima di 40 caratteri.)

time (a) — assegna l'ora corrente alla variabile alfa a

putseg, getseg, e le estensioni e rewrite e reset sono discusse nella sezione 13,A,1.

Funzioni

card (x) — dà la cardinalità dell'insieme x : cioé il numero di elementi contenuti nell'insieme x.

clock — è una funzione, senza parametri, che fornisce un valore intero uguale al tempo di unità centrale, espresso in millisecondi, già usato dal lavoro in esecuzione.

expo (x) — dà l'esponente intero della rappresentazione in virgola mobile del valore reale x; expo (x) = entier (log2(abs(x)))

undefined (x) — funzione booleana. Il suo valore è vero quando il valore reale x è "indefinito" o "oltre i limiti", altrimenti è falso [7].

eos (x) — (discussa nella sezione 13, A, 1)

trunc (x, n) — = trunc (x*y), in cui n è un'espressione intera a y = 2**n.

CAPITOLO 14

COME USARE IL SISTEMA PASCAL 6000 - 3.4

A. Direttive di controllo (per SCOPE 3.4)

Un lavoro Pascal è generalmente completato in 4 fasi. Nella prima è caricato il compilatore Pascal. Nella seconda, fase di compilazione, è generato il codice compilato e l'elenco istruzioni. Nella terza fase il codice compilato, memorizzato dal compilatore su memoria secondaria, è caricato ed agganciato con sottoprogrammi per la gestione dell'ingresso e l'uscita che sono forniti da flussi di libreria. Prima di queste quattro fasi bisogna dare le *istruzioni di controllo* appropriate al sistema operativo. La forma esatta di queste istruzioni e la loro forma abbreviata (il caricamento e l'esecuzione possono spesso essere richiesti con un'unica istruzione) dipendono dal sistema operativo disponibile e sono perciò strettamente dipendenti dal tipo di installazione.

I parametri flusso effettivi, che corrispondono agli identificatori formali nell'intestazione del programma, devono essere specificati nell'istruzione che inizia l'esecuzione del programma compilato (di soltito un comando di EXECUT).

Anche il compilatore è un programma Pascal. La sua intestazione è:

program Pascal (input, output, Lgo);

Il primo parametro formale rappresenta il programma sorgente; il secondo, l'elenco delle istruzioni; ed il terzo il codice compilato "binario" e rilocabile.

I sistemi operativi CDC permettono di omettere i parametri effettivi nelle istruzioni di controllo. Se si omette il nome effettivo di un flusso, le convenzioni Pascal sui parametri dei programmi specificano che gli identificatori formali devono essere usati come nomi effettivi. Perciò i flusso standard INPUT, OUTPUT, e LGO sono automaticamente - per difetto - assunti come

flusso sorgente, elenco istruzioni e codice binario rilocabile. Si tenga però presente che queste funzioni possono essere espletate da altri flussi quando il loro nome è dato come parametro effettivo.

Nota: i parametri effettivi possono essere di, al massimo, 7 caratteri.

B. Opzioni del compilatore

Si possono dare dei comandi al compilatore affinché generi un codice in accordo con certe opzioni; in particolare si può richiedere l'omissione di istruzioni di verifica durante l'esecuzione. Le direttive al compilatore sono scritte come commenti: sono caratterizzate da $ come primo carattere:

{ $ <sequenza di opzioni> <qualsiasi commento> }

Esempio: { $T +, P+ }

Le opzioni possono essere una sequenza di ordini separati da virgole. Ogni ordine è costituito da una lettera, che designa l'opzione, seguita da un più (+) se l'opzione deve essere attivata oppure da un meno (-) se l'opzione non va attivata; in certi casi (X e B più avanti) può essere seguito da un numero.

Attualmente sono disponibili le seguenti opzioni:

T include la verifica in esecuzione per:

a) tutte le operazioni sugli indici dei vettori per assicurare che l'indice sia nei limiti specificati.
b) tutti gli assegnamenti a variabili di tipo sottocampo per controllare che i valori siano all'interno del campo specificato.
c) tutte le divisioni per essere sicuri che non si divida per zero.
d) tutte le conversioni automatiche da intero a reale per assicurarsi che i valori convertiti soddisfino:
   abs (i) <= maxint
e) tutte le istruzioni di case per assicurare che il selettore corrisponda ad una delle etichette specificate

per difetto T+

P genera il codice necessario per generare un'immagine post-mortem nel caso di errore in esecuzione (si veda la sezione 14.C.2)

per difetto P+

X Se un numero n (O <= n <= 6) segue X, passa i primi n descrittori dei parametri nei registri da X0 a X (n-1) (il primo in X0; il secondo in X1, ecc.) Altrimenti li passa nelle locazioni con indirizzi da B6+3 a B6+n+2.

n>0 riduce la quantità di codice prodotto dal compilatore e probabilmente lo migliora leggermente. Il programmatore deve, tuttavia, fare attenzione al fatto che con n>0, il compilatore non può usare i registri e da X0 a X min (n-1, i-2) per il passaggio dell'i - mo parametro.
È perciò possibile che per n>o, il compilatore dia il messaggio di "registri esauriti"; con n=0 non può succedere.

per difetto X4

E permette al programmatore di controllare i simboli per i punti di ingresso dei moduli di codice oggetto (procedure e funzioni) che dichiara nel suo programma. Sono adottate le seguenti convenzioni:

-- I moduli dichiarati "extern" o "fortran" prendono come punto di ingresso un nome eguale ai primi 7 caratteri dell'identificatore di procedura.

-- I moduli locali prendono come punto di ingresso un nome che dipende dal valore dell'opzione E.

E- dal compilatore è generato un unico simbolo
E+ sono presi i primi 7 caratteri del nome del modulo.

Tutte le volte che è preso il nome di un modulo (E+ e "extern" oppure "Fortran"), è a carico del programmatore evitare che esistano dei simboli doppi.

per difetto E-

L controlla la generazione dell'elenco delle istruzioni del programma

per difetto L+

U permette all'utente di ridurre il numero di caratteri validi di ogni linea a 72; il resto della linea è considerato commento.
Con U- il numero dei caratteri è 120: il resto è naturalmente commento.

per difetto U-

B è usato per specificare il limite inferiore per le dimensioni dei buffers di flusso. Se dopo B c'é una cifra d (1 <=d<= 9), la dimensione del buffer, calcolata dal compilatore, è sicuramente maggiore di 128 d parole.

per difetto B1

Dal momento che le direttive al compilatore possono essere scritte in qualunque punto del programma, è possibile attivare le opzioni in modo selettivo su particolari pezzi di programma.

C. Messaggi di errore

C.1 Compilatore

Ogni errore, individuato dal compilatore, è segnalato con una freccia che punta al testo errato, seguita da un numero che corrisponde ai messaggi nell'appendice E.

C.2 In esecuzione (immagine post-mortem)

Quando è abilitata l'opzione P (P+), il compilatore genera del codice che può essere usato per generare, nel caso di errore durante l'esecuzione, una "immagine leggibile."
L'immagine comprende le seguenti opzioni:

a) la causa della trappola ed il punto di innesco

b) una descrizione di quelle procedure (funzioni) attivate al momento della trappola. La descrizione, in ordine inverso rispetto alla chiamata, è costituita da:
1) il nome della procedura
2) il punto di chiamata
un elenco di nomi delle variabili e parametri locali

c) i valori delle variabili globali nel programma principale.

Sono elencate solo le variabili ed i parametri di tipo intero, reale, booleano e carattere. I puntatori o sono "nil" oppure hanno un valore ottale (indirizzo). Per le altre variabili scalari, è stampato il numero ordinale del loro valore corrente. Quando, per una qualunque proceudra, l'opzione P non è attivata (P-) viene generato solo il nome della procedura e la locazione di chiamata.

Nel caso di chiamate recursive, sono elencate solo gli ultimi tre inneschi di ogni procedura (i 3 più recenti).

RIFERIMENTI

1. N. Wirth, *The programming Language PASC..., Acta Informatica, 1, 35-63, 1971.*

2. ----, *"Program Development by Stepwise Refinement", Comm. ACM 14*, 221-227, April 1971.

3. ----, *Systematic Programming*, Prentice-Hall, Inc. 1973.

4. O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare, *Structured Programming*, Academic Press Inc. 1972.

5. C.A.R. Hoare and N. Wirth, "An Axiomatic Definition if the Programming Language Pascal", *Acta Informatica,* 2, 335-355, 1973.

6. D.E. Knuth,l THE ART OF COMPUTER PROGRAMMING, vol 1, *Fundamental Algorithms,* Addison-Wesley, 1968.

7. SCOPE Reference Manual, CDC 6000 Version 3.4.1, Control Data Corporation, 1973.

8. N. Wirth "The Design of a Pascal Compiler", SOFTWARE -Practice and Experience, 1, 309-333 (1971)

APPENDICE A

PROCEDURE E FUNZIONI, STANDARD

Procedure per il trattamento dei flussi

put (f) appende, al flusso f, il valore della variabile buffer f↑; è applicabile solo se prima dell'esecuzione eof (f) è vero; eof (f) rimane vero e f↑ diviene indefinito.

get (f) posiziona sulla prossima componente del flusso ed assegna questa componente alla variabile buffer f↑. Se non esiste una "prossima componente", eof (f) diventa vero ed il valore di f↑ è indefinito. È applicabile solo se eof (f) è falso prima dell'esecuzione.

reset (f) riposiziona, per permettere una lettura, all'inizio del flusso la posizione corrente: assegna cioé alla variabile buffer f↑ il valore del primo elemento di f; eof (f) diviene falso se f non è vuoto, altrimentio f↑ è indefinito ed eof (f) rimane vero.

rewrite (f) sostituisce il valore corrente di f con il flusso vuoto. eof (f) diventa vero e può essere scritto con nuovo flusso.

page (f) fa sì che la stampante passi all'inizio della nuova pagina prima di stampare la prossima linea di un flusso-testo.

read, readln, write, writeln, sono analizzate nel capitolo 12.

Procedure di allocazione dinamica

new (p) alloca una nuova variabile v ed assegna l'indirizzo di v alla variabile puntatore p. Se v è tipo record con variante, la forma

new (p, t1,...tn) può essere usata per allocare una variabile con valore di campo discriminatore t1...tn. I valori dei campi discriminatori devono essere elencati in modo contiguo e nello stesso ordine della loro dichiarazione; non possono essere cambiati durante l'esecuzione.

dispose (p) indica che la memoria occupata dalla variabile p non è più necessaria.

dispose (p, t1,...,tn) può essere usata per indicare che memoria occupata dalla variabile p (con valore di campo discriminatore t1...tn) non è più necessaria. I valori dei campi devono essere identici a quelli usati nell'allocazione della variabile.

Procedure per trasformazione dati

pack (a, i, z) se a è una variabile vettore del tipo
array [m..n] **of** T
e z è una variabile di tipo
packed array [u..v] **of** T
in cui n-m >= v-u, allora la procedura è equivalente a:

for j := u **to** v **do** z [j] := a [j-u+i]

unpack (z, a, i) è equivalente a

for j := u **to** v **do** a [j-u+i] := z [j]

Funzioni aritmetiche

abs (x) calcola il valore assoluto di x. X può essere sia intero che reale ed il risultato è dello stesso tipo di x.

sqr (x) calcola x*x. X può essere sia intero che reale ed il risultato è dello stesso tipo di x.

sin (x) mentre il risultato è sempre reale, x può essere intero o reale (Questo vale anche per le funzioni che seguono).

cos (x)
arctan (x)
exp (x)
ln (x) (logaritmo naturale)
sqrt (x) (radice quadrata)

Predicati (funzioni booleane)

odd (x) x deve essere intero; il risultato è vero se x è dispari e falso se è pari.

eoln (f) dà il valore vero quando, leggendo il flusso-testo f, è raggiunta la fine della linea corrente.

eof (f) dà il valore vero quando, leggendo il flusso-testo f, è raggiunta la fine del flusso.

Funzioni di trasformazione

trunc (x) x deve essere reale; il risultato è l'intero più grande minore od eguale ad x per x>=0, ed il più piccolo intero maggiore od eguale ad x per x<=0.

round (x) x deve essere di tipo reale; il risultato - intero è il valore arrotondato. Vale a dire:

$$\text{round}(x) = \begin{cases} \text{trunc}(x+0.5), & \text{per } x \geqslant 0 \\ \text{trunc}(x-0.5), & \text{per } x < 0 \end{cases}$$

ord (x) dà il numero associato ad x nell'insieme dei valori definiti dal tipo x.

chr (x) x deve essere intero ed il risultato è il carattere il cui numero ordinale è x (se esiste).

Altre funzioni standard

succ (x) x può essere di qualunque tipo scalare, eccettuato reale, ed il risultato è il successore di x (se questi esiste)

pred (x) x può essere di qualunque tipo scalare, eccettuato reale, ed il risultato è il predecessore di x (se questi esiste).

APPENDICE B

SOMMARIO DEGLI OPERATORI

Operatore	*Operazione*	*Tipo operanai*	*Tipo risultato*
:=	assegnamento	qualunque tipo eccettuati flussi	---
aritmetici:			
+ (unario)	identità	intero o reale	stesso dell'operando
- (unario)	inversione di segno		
+	addizione	intero o reale	intero o reale
-	sottrazione		
*	moltiplicazione		
div	divisione intera	intero	intero
/	divisione reale	intero o reale	reale
mod	modulo	intero	intero
relazioni:			
=	uguaglianza	scalare, stringa, set o puntatore	Booleano
<>	disuguaglianza		
<	minore di	scalare o stringa	
>	maggiore di		
<=	minore o uguale -o- inclusione	scalare o stringa set	
>=	maggiore o uguale -o- inclusione	scalare o stringa set	
in	appartenenza	primo operando scalare, il secondo è il suo tipo set	
logici:			
not	negazione	Booleano	Booleano
or	disgiunzione		
and	congiunzione		
su insiemi:			
+	unione	qualsiasi ins. tipo T	T
-	differenza di insiemi		
*	intersezione		

APPENDICE C

TAVOLE

A. Identificatori standard

Costanti:
false, true, maxint

Tipi:
integer, Boolean, real, char, text

Parametri di programma:
input, output

Funzioni:
abs, arctan, chr, cos, eof, eoln, exp, ln, odd, ord, pred, round, sin, sqr, sqrt, succ, trunc

Procedure:
get, new, pack, page, put, read, readln, reset, rewrite, unpack, write, writeln

B. Parole delimitatrici (parole riservate)

and	**end**	**nil**	**set**
array	**file**	**not**	**then**
begin	**for**	**of**	**to**
case	**function**	**or**	**type**
const	**goto**	**packed**	**until**
div	**if**	**procedure**	**var**
do	**in**	**program**	**while**
downto	**label**	**record**	**with**
else	**mod**	**repeat**	

C. Identificatori non standard predefiniti in PASCAL 6000 - 3.4

Tipi:
alfa

Funzioni:
card, clock, eos, expo, undefined

Procedure:
date, getseg, halt, linelimit, message, putseg, time

APPENDICE D

SINTASSI

Formalismo di Backus-Naur (BNF)

Si noti che i simboli

::= | { }

non sono simboli del linguaggio PASCAL ma dei meta-simboli del formalismo BNF. Le parentesi graffe stanno ad indicare 0 o più ripetizioni dei simboli racchiusi; quindi:

A ::= { B }

è una formulazione sintetica della regola puramente recursiva:

A ::= <vuoto> | AB

<programma> ::= <intestazione programma> <blocco>
<intestazione programma> ::= **program** <identificatore>
(<identificatore flusso>
{, <identificatore flusso> });
<identificatore flusso> ::= <identificatore>
<identificatore> ::= <lettera> { <lettera o cifra> }
{lettera o cifra> ::= <lettera> | <cifra>
{blocco} ::= <parte dichiarazione etichette>
<parte definizione costanti> <parte definizione tipo>
<parte dichiarazione variabili>
<parte dichiarazione funzioni e procedure> <parte istruzioni>
<parte dichiarazione etichette> ::= <vuoto> | **label** <etichetta>
{, <etichetta> };
<vuoto> ::=
<etichetta> ::= intero senza segno>
<parte definizioni costanti> ::= <vuoto> | **const** <definizione costanti>
{, <definizione costanti> };
<definizione costanti> ::= <identificatore> = <costante>
<costante> ::= <numero senza segno> | <segno>
<numero senza segno> | <identificatore costante> |
<segno> <identificatore costante> | <stringa>
<numero senza segno> ::= <intero senza segno> | <reale senza segno>
<intero senza segno> ::= <cifra> { <cifra> }

<reale senza segno> ::= <intero senza segno> . <cifra> { <cifra> } |
<intero senza segno> . <cifra> { <cifra> } E
<fattore di scala> |
<intero senza segno> E <fattore di scala>

<fattore di scala> ::= <intero senza segno> | <segno>
<intero senza segno>

<segno>::= + | -
<identificatore costante> ::= <identificatore>
<stringa> ::= ' <carattere> { <carattere> } '
<parte definizione tipo> ::= <vuoto> | **type** <definizione tipo>
{; <definizione tipo> };

<definizione tipo> ::= <identificatore> = <tipo>
<tipo> ::= <tipo semplice> | <tipo strutturato> | <tipo puntatore>
<tipo semplice> ::= <tipo scalare> | <tipo sottocampo> |
<identificatore tipo>

<tipo scalare> ::= (<identificatore> {, <identificatore> })
<tipo sottocampo> ::= <costante> .. <costante>
<identificatore tipo> ::= <identificatore>
<tipo strutturato> ::= <tipo strutturato disimpaccato> |
packed <tipo strutturato disimpaccato>

<tipo strutturato disimpaccato> ::= <tipo vettore> | <tipo record> |
<tipo set> | <tipo flusso>
<tipo vettore> ::= **array** [<tipo indice> {, <tipo indice> }] **of**
<tipo componente>

<tipo indice> ::= <tipo semplice>
<tipo componente> ::= <tipo>
<tipo record> ::= **record** <elenco campi> **end**
<elenco campi> ::= <parte fissa> | <parte fissa> ; <parte variante> |
<parte variante>
<parte fissa> ::= <sezione record> {; <sezione record> }
<sezione record> ::= <identificatore campo>
{, <identificatore campo> } : <tipo> | <vuoto>
<parte variante> ::= **case** <campo discriminatore> <identificatore tipo>
of <variante> {; <variante> }
<campo discriminatore> ::= <identificatore campo> : | <vuoto>
<variante> ::= <elenco etichette case> : (<elenco campi>) | <vuoto>
<elenco etichette case> ::= <etichette case> {, <etichetta case> }
<etichetta case> ::= <costante>
<tipo set> ::= **set of** <tipo base>
<tipo base> ::= <tipo semplice>
<tipo flusso> ::= **file of** <tipo>
<tipo puntatore> ::= ↑ <identificatore tipo>

<parte dichiarazione variabili> ::= <vuoto> | **var**
<dichiarazione variabile>
{; <dichiarazione variabile> };
<dichiarazione variabile> ::= <identificatore> {, {identificatore> } :
<tipo>
<parte dichiarazione procedure o funzioni>> ::= <dichiar. proc. o funz.>

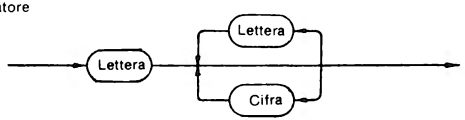
<dichiarazione procedura o funzione> ::= <dichiarazione procedura>|
<dichiarazione funzione>
<dichiarazione procedura> ::= <intestazione procedura> <blocco>
<intestazione procedura> ::= **procedure** <identificatore>; |
procedure <identificatore>
(<sezione parametro formale>
{; <sezione parametro formale> });
<sezione parametro formale> ::= <gruppo parametro> | **var**
<gruppo parametro> | **function**
<gruppo parametro> | **procedure** <identificatore>
{, <identificatore> }
<gruppo parametro> ::= identificatore> {, <identificatore> } :
<identificatore tipo>
<dichiarazione funzione> ::= intestazione funzione> <blocco>
<intestazione funzione> ::= **function** <identificatore> : <tipo risultato> ;
function <identificatore> (<sezione parametro
formale> {; <sezione parametro> }): <tipo risultato> ;
<tipo risultato> ::= <identificatore tipo>
<parte istruzioni> ::= <istruzione composta>
<istruzione composta> ::= **begin** <istruzione> {; istruzione } **end**
<istruzione> ::= <istruzione senza etichetta> | <etichette> :
<istruzione senza etichette>
<istruzione senza etichette> ::= <istruzione semplice> |
<istruzione strutturata>
<istruzione semplice> ::= <assegnamento> | <chiamata di procedura>
| <go to> | <vuoto>
<assegnamento> ::= <variabile> := <espressione> |
<identificatore funzione> := <espressione>
<variabile> ::= <variabile semplice> <variabile componente>
<variabile indirizzata>
<variabile semplice> ::= <identificatore variabile>
<identificatore variabile> ::= <identificatore>
<variabile componente> ::= <variabile indicizzata> |
<designatore campo> | <buffer flusso>
<variabile indicizzata> ::= <variabile vettore>
[<espressione> {, <espressione> }]

<variabile vettore> ::= <variabile>
<designatore campo> ::= <variabile record> . <identificatore campo>
<variabile record> ::= <variabile>
<identificatore campo> ::= <identificatore>
<buffer flusso> ::= <variabile flusso> ↑
<variabile flusso> ::= <variabile>
<variabile indirizzata> ::= <variabile puntatore> ↑
<variabile puntatore> ::= <variabile>
<espressione> ::= <espressione semplice> | <espressione semplice>
<operatore relazionale>
<espressione semplice>
<operatore relazionale> ::= = | <> | < | <= | >= | > | **in**
<espressione semplice> ::= <termine> | <segno> <termine> |
<espressione semplice> <operatore additivo>
<termine>
<operatore addittivo> ::= + | - | **or**
<termine> ::= <fattore> | <termine> <operatore multiplicativo>
<fattore>
<operatore multiplicativo> ::= * | / | **div** | **mod** | **and**
<fattore> ::= <variabile> | <costante senza segno> |
(<espressione>) | <designatore funzionale> |
<set> | **not** <fattore>
<costante senza segno> ::= <numero senza segno> | <stringa> |
| <identificatore costante> | **nil**

<designatore funzionale> ::= <identificatore di funzione> |
<identificatore di funzione>
(<parametro effettivo>
{, <parametro effettivo> })
<identificatore di funzione> ::= <identificatore>
<set>::= [<elenco elementi>]
<elenco elementi> ::= <elemento> . {, <elemento> } | vuoto
<elemento> ::= <espressione> | <espressione> .. <espressione>
<chiamata di procedura> ::= <identificatore procedura> |
<identificatore procedura>
(parametro effettivo
{, <parametro effettivo> })
<identificatore procedura> ::= <identificatore>
<parametro effettivo> ::= <espressione> | <variabile> |
<identificatore procedura> |
<identificatore funzione>
<go to> ::= **goto** <etichetta>

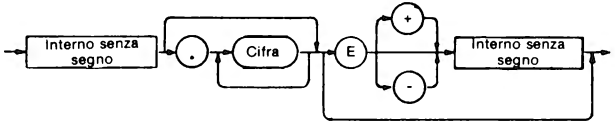
<istruzione strutturata> ::= <istruzione composta> |
<istruzione condizionale> |
<istruzione ripetitiva> |
<istruzione with>
<istruzione condizionale> ::= <istruzione if> | <istruzione selezione>
<istruzione if> ::= **if** <espressione> **then** <istruzione> | **if** <espressione>
then <istruzione> **else** <istruzione>
<istruzione selezione> ::= **case** <espressione> **of**
<elemento elenco selezione>
{; <elemento elenco selezioni> } **end**
<elemento elenco selezione> ::= <elenco etichette selezione> :
<istruzione> | <vuoto>
<elenco etichette selezione> ::= <etichetta selezione>
{, <etichetta selezione> }
<istruzione ripetitiva> ::= <istruzione while> | <istruzione repeat> |
<istruzione for>
<istruzione while> ::= **while** <espressione> **do** <istruzione>
<istruzione repeat> ::= **repeat** <istruzione> {; <istruzione> }
until <espressione>
<istruzione for> ::= **for** <variabile controllo> := <elenco for> **do**
<istruzione>
<elenco for> ::= <valore iniziale> **to** <valore finale> |
<valore iniziale> **downto** <valore fianale>
<variabile controllo> ::= <identificatore>
<valore iniziale> ::= <espressione>
<valore finale> ::= <espressione>
<istruzione with> ::= **with** <elenco variabili record> **do** <istruzione>
<elenco variabili record> ::= <variabile record> {, <variabile record> }

Identificatore

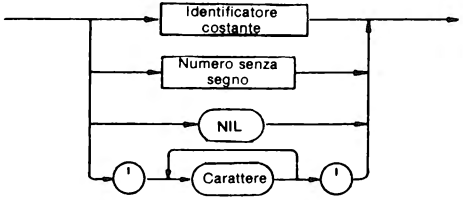


Interno senza segno

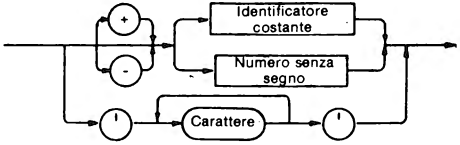
Numero senza segno



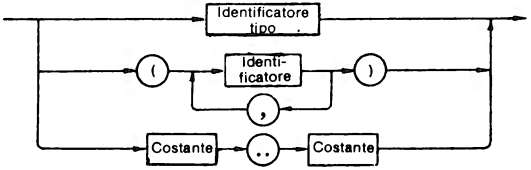
Costante senza segno



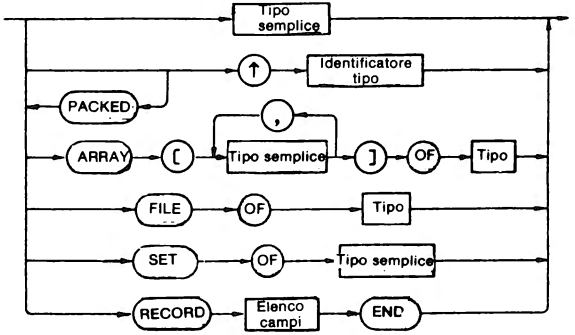
Costante



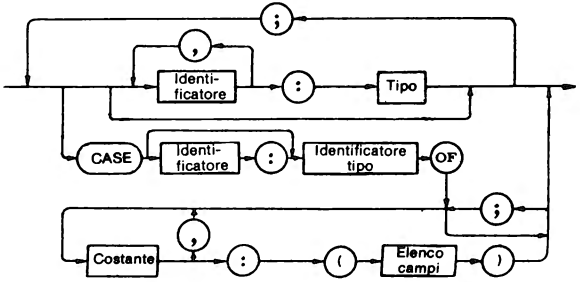
Tipo semplice



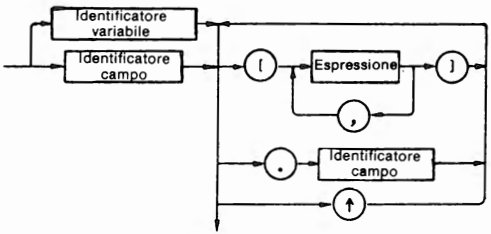
Tipo



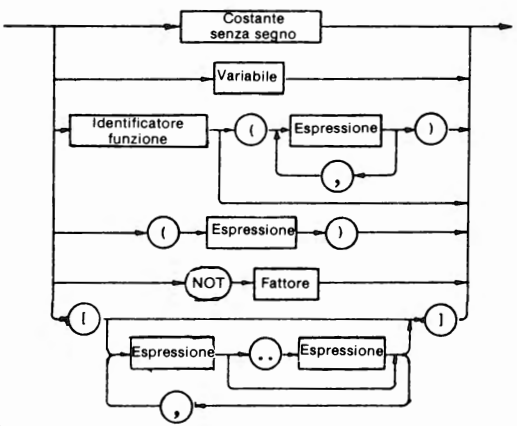
Elenco campi



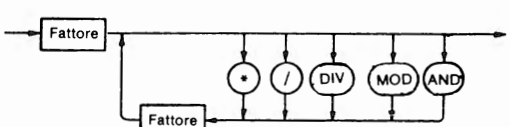
Variabile



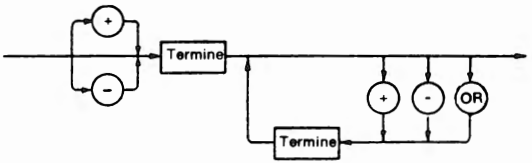
Fattore



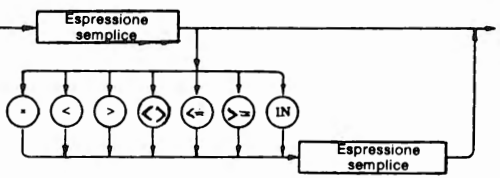
Termine



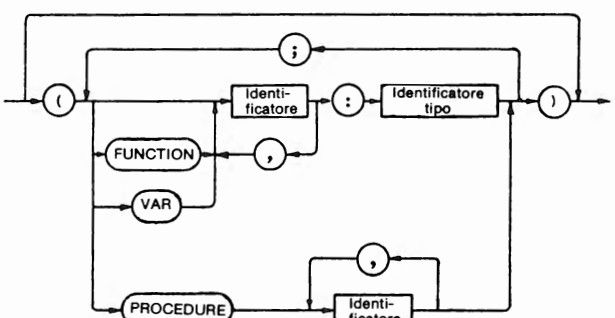
Espressione semplice



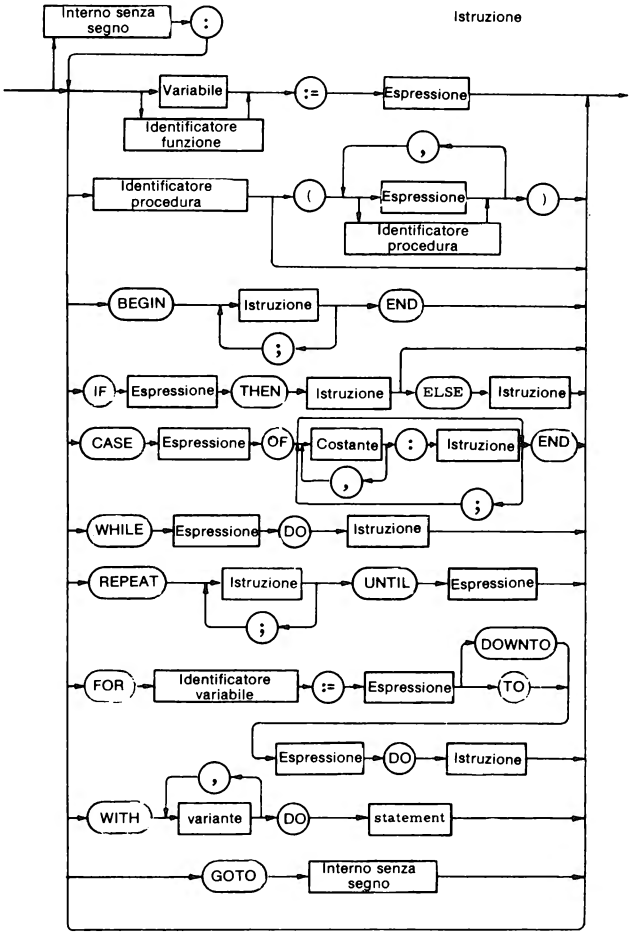
Espressione



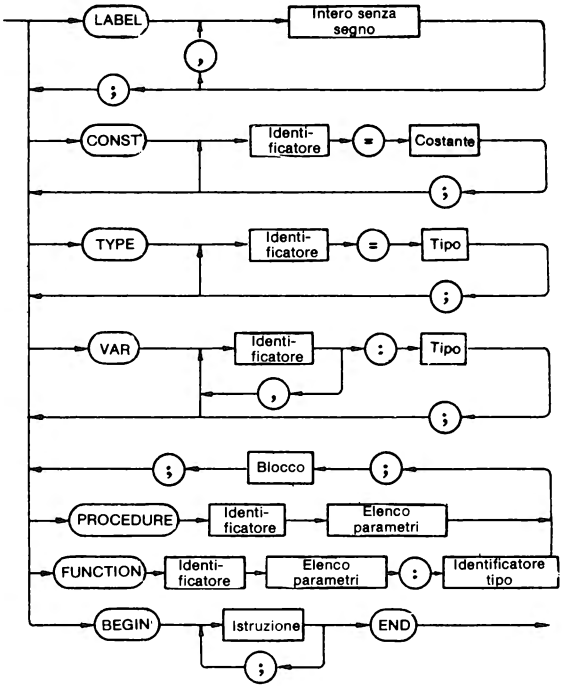
Elenco parametri



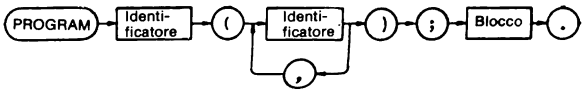
Istruzione



Blocco



Programma



APPENDICE E

NUMERAZIONE ERRORI

1 : errore su tipo semplice
2 : è richiesto un identificatore
3 : è richiesto 'program'
4 : è richiesto ')'
5 : è richiesto ':'
6 : simbolo non permesso
7 : errore nell'elenco parametri
8 : è richiesto 'of'
9 : è richiesto '('
10 : errore su tipo
11 : è richiesto ' ['
12 : è richiesto ']'
13 : è richiesto 'end'
14 : è richiesto ';'
15 : è richiesto un intero
16 : è richiesto '='
17 : è richiesto 'begin'
18 : errore nella sezione dichiarazione
19 : errore nell'elenco campi
20 : è richiesto '.'
21 : è richiesto ' * '

50 : errore in una costante
51 : è richiesto ':='
52 : è richiesto 'then'
53 : è richiesto 'until'
54 : è richiesto 'do'
55 : è richiesto 'to' / 'downto'
56 : è richiesto 'if'
57 : è richiesto 'file'
58 : errore di fattore
59 : errore di variabile

101 : identificatore dichiarato due volte
102 : limite inferiore superiore al limite superiore
103 : identificatore non della classe corretta
104 : identificatore non dichiarato

105 : non è permesso il segno
106 : è richiesto un numero
107 : tipi sottocampo incompatibili
108 : qui non è concesso un flusso
109 : il tipo non deve essere reale
110 : il campo discriminatore deve essere scalare o sottocampo
111 : non compatibile con il campo discriminatore
112 : l'indice non può essere reale
113 : l'indice deve essere scalare o sottocampo
114 : il tipo base non deve essere reale
115 : il tipo base deve essere scalare o sottocampo
116 : il tipo del parametro di una procedura standard è errato
117 : riferimento in avanti non soddisfatto
118 : riferimento in avanti per identificatore tipo in dichiarazione variabile
119 : dichiarato in "avanti"; non è concesso ripetere elenco parametri
120 : il tipo del risultato di una funzione deve essere scalare, sottocampo o puntatore
121 : non è permesso un valore flusso come parametro
122 : funzione dichiarata in avanti; non è permessa la ripetizione del tipo del risultato
123 : è stato omesso il tipo del risultato nella dichiarazione di funzione
124 : formato F solo per reali
125 : è errato il tipo di un parametro di una funzione standard
126 : il numero dei parametri non si accorda con la dichiarazione
127 : sostituzione non legale di parametri
128 : non c'é accordo tra dichiarazione ed il tipo del risultato di un parametro funzione
129 : conflitto tra il tipo degli operandi
130 : l'espressione non è di tipo set
131 : è concessa solo la verifica di disuguaglianza
132 : non è concessa inclusione stretta
133 : non è concesso confronto di flussi
134 : il tipo degli operandi non è tra quelli consentiti
135 : il tipo degli operandi deve essere booleano
136 : il tipo degli elementi di un insieme deve essere scalare o sottocampo
137 : i tipi degli elementi di un insieme non sono compatibili
138 : la variabile non è di tipo vettore
139 : il tipo dell'indice non è compatibile con la dichiarazione
140 : la variabile non è di tipo record
141 : la variabile deve essere di tipo flusso o puntatore
142 : sostituzione non legale di parametri
143 : la variazione di controllo della ripetizione è di tipo non consentito
144 : espressione di tipo non consentito
145 : conflitto di tipi

146 : non è concesso l'assegnamento a flussi
147 : il tipo dell'etichetta è incompatibile con l'espressione di selezione
148 : i limiti di un sottocampo devono essere scalari
149 : l'indice non può essere di tipo intero
150 : non è concesso l'assegnamento a funzioni standard
151 : non è concesso l'assegnamento a funzioni formali
152 : non c'é un campo come questo nel record
153 : errore di tipo in lettura
154 : il parametro effettivo deve essere una variabile
155 : la variabile di controllo non deve essere dichiarata in un livello intermedio
156 : definizione multipla di etichetta in una selezione
157 : troppi casi in un'istruzione di selezione
158 : omessa la dichiarazione della variante corrispondente
159 : non sono permessi reali o stringhe come campi selezionatori
160 : la dichiarazione precedente non era "in avanti"
161 : ancora dichiarata in avanti
162 : la dimensione del parametro deve essere costante
163 : variante omessa nella dichiarazione
164 : non è connessa la sostituzione di proc/funzioni standard
165 : etichetta con definizioni multiple
166 : etichetta con dichiarazioni multiple
167 : etichetta non dichiarata
168 : etichetta non definita
169 : errore nell'insieme base
170 : è richiesto un parametro valore
171 : flusso standard ri-dichiarato
172 : flusso esterno non dichiarato
173 : è richiesta una procedura o funzione fortran
174 : è richiesta una procedura o funzione pascal
175 : omesso il flusso "input" nell'intestazione del programma
176 : omesso il flusso "output" nell'intestazione del programma
177 : qui non sono concessi assegnamenti a identificatori di funzione
178 : variante di record con definizioni multiple
179 : X-opt di proc/funz. effettive non in accordo con dichiarazione formale
180 : la variabile di controllo non deve essere formale
181 : la parte costante indirizzo fuori dai limiti

201 : errore in costante reale: è richiesta cifra
202 : una costante stringa non deve superare una linea
203 : costante intera oltre i limiti
204 : 8 o 9 in numero ottale

205 : ci deve essere almeno una stringa
206 : parte intera di una costante reale oltre i limiti

250 : troppe nidificazioni di identificatori
251 : troppe nidificazioni di procedure e/o funzioni
252 : troppi riferimenti in avanti di ingressi di procedure
253 : procedura troppo lunga
254 : costanti troppo lunghe in questa procedura
255 : troppi errori in questa linea sorgente
256 : troppi riferimenti esterni
257 : troppi esterni
258 : troppi flussi locali
259 : espressione troppo complicata
260 : troppe etichette d'uscita

300 : divisione per zero
301 : non è previsto un caso per questo valore
302 : espressione dell'indice oltre i limiti
303 : il livello che va assegnato è oltre i limiti
304 : elemento dell'espressione non nei limiti

398 : restrizione implementativa
399 : non sono implementate dimensioni variabili per vettori

APPENDICE F

ESEMPI DI PROGRAMMAZIONE

```
{procedure per leggere e scrivere dei numeri reali usate dalle procedure stan-
dard read (f, x) e write (f, x:n)                                          }

procedure rdr (var f: text; var x: real);
    { legge i numeri reali in formato libero }
    const t4 8 = 28147976710656;
              limit = 56294995342131;
              z = 27; { ord ('0') }
              liml = 322; { esponente massimo }
              lim2 = -292; { esponente minimo }
    type posint = 0..323;
    var ch: char; y: real; a, i, e: integer;
        s, ss: boolean { segni }
    function ten (e: posint): real; { = 10**e, 0<e <322 }
            var i: integer; t: real;
    begin i := 0; t := 1.0
            repeat if odd (e) then
                    case i of
                        0 : t := t * 1.0e1;
                        1 : t := t * 1.0e2;
                        2 : t := t * 1.0e4;
                        3 : t := t * 1.0e8;
                        4 : t := t * 1.0e16;
                        5 : t := t * 1.0e32;
                        6 : t := t * 1.0e64;
                        7 : t := t * 1.0e128;
                        8 : t := t * 1.0e256;
                    end;
                    e := e div 2; i := i+1
            until e=0
            ten := t
    end;

begin
    { superametro spazi in testa }
            while f ↑ = ' ' do get (f);
            ch := f ↑;
```

```
if ch = '-' then
    begin s := true; get (f); ch := f ↑
    end else
    begin s := false;
            if ch = '+' then
            begin get (f); ch := f ↑
            end
    end;

    if not (ch in ['0' .. '9'] then
    begin message ('** richiesta cifra); halt;
    end;
    a := 0; e := 0;
    repeat if a <limit then a := 10 * a + ord (ch) -z
                        else   e := e+1;
            get (f); ch := f ↑
    until not (ch in ['0' .. '9'] );
    if ch = '.' then
    begin {lettura frazione} get (f); ch := f ↑;
        while ch in ['0' .. '9'] do
        begin if a <limit then
                begin a := 10*a + ord (ch) -z; e := e-1
                end;
        get (f); ch := f ↑
      end
    end;

    if ch = 'e' then
    begin {lattura fattore scala} get (f); ch := f ↑;
        i := 0;
        if ch = '-' then
        begin ss := true; get (f); ch := f ↑
        end else
        begin ss := false; if ch = '+' then
            begin get (f); ch := f ↑
            end

        end;

        if ch in ['0' .. '9'] then
        begin i := ord (ch) -z; get (f); ch := f ↑;
            while ch in ['0' .. '9'] do
            begin if i < limit then i := 10*i + ord (ch) -z;
                get (f); ch := f ↑
            end
```

```
            end else
            begin message ('richiesta cifra'); halt
            end;
            if ss then e := e-i else e := e+i;
        end;
        if e < lim2 then
            begin a := 0; e := 0
            end else
        if e > liml then
        begin message ('** numero troppo grande'); halt end;
        { 0<a < 2**49 }
        if a > = t48 then y := ((a+1) div 2)* 2.0
                    else y := a;
        if s then x := -y;
        if e < 0 then x := y/ten (-e) else
        if e <> 0 then x := *ten (e) else x := y;
    end;

procedure wre (var f : text; x : real; n : integer);
    { scrive un numero reale x con n caratteri nel formato decimale virgola
    mobile. Le costanti che seguono sono vincoli del CDC. }
    const t48 = 28147497710656; = 2**48
          z = 27; {ord ('0')}
    type posint = 0..323;
    var c, d, e, e0, e1, e2, i : integer;
    function ten (e:posint) : real; {10**e, 0 < e < 322}
        var i : integer; t : real;
    begin i := 0; t := 1.0.
        repeat if odd(e) then
            case i of
                0 : t := t * 1.0e1;
                1 : t := t * 1.0e2;
                2 : t := t * 1.0e4;
                3 : t := t * 1.0e8;
                4 : t := t * 1.0e16;
                5 : t := t * 1.0e32;
                6 : t := t * 1.0e64;
                7 : t := t * 1.0e128;
                8 : t := t * 1.0e256;
            end;
            e := e div 2 ; i := i+1
        until e=0;
        ten := t
    end {ten};
```

```
begin {necessari almeno 10 caratteri: b+9.9e+999}
    if undefined (x) then
    begin repeat f ↑ := ' '; put (f); n := n-1
          until n < =1;
         f ↑ := 'n'; put (f)
    end else
    if x=0 then
    begin repeat f ↑ := ' '; put (f); n := n-1
          until n < =1;
         f ↑ := 'u'; put (f)
    end else
         begin
         if n < = 10 then n := 3 else n := n-7;
         repeat f ↑ := ' '; put (f); n := n-1
         until n < = 15;
         {1 < n < 15 numero di cifre da stampare}
         begin {verificat del segno, prepar. esponente}
              if x < 0 then
                begin f ↑ := '-'; put (f); x := -x
                end else begin f ↑ := ' '; put (f) end;
              e := expo (x);
              if e < = 0 then
                begin e := e*77 div 256+1; x := x/ten (e);
                     if x > = 1.0 then
                       begin x := x/100; e := e+1
                       end
                end else
                begin e := (e+1)* 77 div 256; x := x/ten (-e)* x;
                     if x < 0.1 then
```

```
                    begin x := 10.0 * x; e := e-1
                    end
    end;
{0.1 < = x < 1.0}
    case n of {arrotondamento}

                 2 : x := x+0.5e-2;
                 3 : x := x+0.5e-3;
                 4 : x := x+0.5e-4;
                 5 : x := x+0.5e-5;
                 6 : x := x+0.5e-6;
                 7 : x := x+0.5e-7;
                 8 : x := x+0.5e-8;
                 9 : x := x+0.5e-9;
                10 : x := x+0.5e-10;
                11 : x := x+0.5e-11;
                12 : x := x+0.5e-12;
                13 : x := x+0.5e-13;
                14 : x := x+0.5e-14;
                15 : x := x+0.5e-15;
            end;
            if x > = 10 then
              begin x := x * 0.1; e := e+1;
              end;
            c := trunc (x, 48);
            c := 10 * c; d := c div t48;
            f ↑ := chr (d+z); put (f);
            f ↑ := '.'; put (f);
            for i := 2 to n do
            begin c := (c-d * t48)* 10; d := c div t48;
                f ↑ := chr (d+z); put (f);
            end;
            f ↑ := 'e'; put (f); e := e-1;
            if e < 0 then
                begin f ↑ := '-'; put (f); e := -e;
                end else begin f ↑ := '+'; put (f); end;
            e1 := e * 205 div 2048; e2 := e -10 * e1;
            e0 := e1 * 205 div 2048; e := e1 -10 * e0;
            f↑:= chr (e0+z); put (f);
            f↑ := chr (e1+z); put (f);
            f↑ := chr (e2+z); put (f);
        end
      end
    end {wre};
```

INDICE ANALITICO

Fatta eccezione per i nomi di sezione (p.e. Appendice A), i riferimenti possono avere una delle forme seguenti:

x1 x1.x2 x1.x2.x3

in cui:

x1 indica il numero del capitolo
x2 fa riferimento ad un paragrafo se è una lettura maiuscola;
ad una figura se è una lettera minuscola;
ad un programma se è un numero
x3 indica un sottoparagrafo

Albero binario 11.A
Alfa (PASCAL 6000-3.4) 13.D.1
Ambiente 0
Blocco 0
Booleano 2.A
Carattere 2.D
Carte sintattiche Appendice D
Commento 1.
Definizioni BNF Appendice D
Effetto margine 11.B
Elenco campi 7.
Equivalenza 2.A
Espressione 4.A
Etichette
- Case 7.A
- Goto 3.B,4.E
- Parte dichiarazione 3.B

Figure
- Carta sintattica della struttura del programma 0.a
- Dopo (inserzione liste) 10.c
- Due persone campione 7.a
- Espressioni 11.a
- Identificatore 1.a
- Insieme caratteri ASCII (con ordinamento CDC) 13.a
- Insieme dei caratteri CDC scientifico (con 64 elementi) 13.a
- Lista concatenata 10.a
- Numero senza segno 1.b

Prima (inserzione lista)11.b
Rappresentazione alternativa dei simboli standard 13.b
Struttura albero binario 11.b
Struttura blocco 0.b
Funzioni
Designatore 3.F
Intestazione 11.B
Predefinite (PASCAL 6000-3.4) 13.D.2
Standard (tabella delle) Appendice A

Identificatori standard Appendice C
Identificatori (tabella degli) Appendice C
Implicazione 2.A
Input 9.B
Insiemi carattere 13.B.3
Intero 2.B
I/O 12.
Istruzioni

Assegnamento 4.A
Case 4.D.2
Composta 4.B
Condizionali 4.D
Controllo (PASCAL 6000-3.4) 14.A
For 4.C.3
Goto 4.E
If 4.D.1
Iterative 4.C
Repeat 4.C.2
Vuota 4.B
While 4.C.1
With 7.A

Liste (concatenate) 10.
Messaggi d'errore del compilatore 14.C.1 Appendice E
Messaggi d'errore durante l'esecuzione 14.C.2
Notazione 1.
Numeri 1.
Operatore relazionale 2.A,4.A
Operatori (sommario degli) Appendice B
Opzioni compilatore (PASCAL 6000-3.4) 13.B
Output 9.B
Parametri 11.A
Parole riservate (tabella delle) Appendice C
Parte dichiarazione 3.
Parte dichiarazione costanti 3.C

PASCAL 6000-3.4 13,14
Precedenza nomi 11.A
Precedenza operatori 4.A
Procedure 11.A

Esterne (PASCAL 6000-3.4) 13.A.2
Intestazione 11.A
Istruzione 11.A
Parte dichiarazione 3.F
Predefinite (PASCAL 6000-3.4) 13.D.2
Read 12.A
Standard (tabella delle) Appendice A
Write 12.B

Programmi

Attraversamento 11.5
Beginend 4.1
Bisez 11.6
Complessi 7.1
Cont freq 9.1
Conversione 3.1
Coseno 4.5
Effetto marg 11.7
Elevpot 4.8
Esalfa 13.1
Espon2 11.8
Gcdrecursivo 11.9
Grafico 1 4.9
Grafico 2 6.2
Inflazione 0.1
Inserzione 9.2
Minmass 6.1
Minmass 2 11.1
Minmass 3 11.2
Moltmatr 6.3
Parametri 11.3
Perfor 4.4
Perrepeat 4.3
Perwile 4.2
Postfix 11.4
Prumprimi 8.2
Romani 4.7
Setop 8.1
Somma 4.6

Reale 2.C
Retrizioni (PASCAL 6000-3.4) 13.C
Riferimento in avanti 11.C

Schemi
- Lettura di un numero arbitrario di numeri da un flusso-testo 12.A
- Lettura di un testo 9.A
- Lettura di un testo da "input" 9.A
- Lettura e scrittura di un flusso segmentato 13.A.1
- Scritture di un flusso segmentato 13.A.1
- Scritture di un testo 9.A
- Scritture di un testo X in Y 9.A

Separatori 1.
Stringa 1,6
Strutture impaccate 6.

Tabelle
- Caratteri di controllo della stampante 9.B,13.B.4
- Operazioni su flussi-testo 9.A
- Simboli speciali 1.
- Struttura blocco 0.
- Valori per difetto della lunghezza campo 13.B.4

Tipo
- Di dato 2.
- Flussi esterni (PASCAL 6000-3.4) 13.B.1
- Flussi segmentati (PASCAL 6000-3.4) 13.A.1
- Flussi-testo 9.A
- Flusso 9.
- Parte dichiarazione 3.D
- Predefiniti (PASCAL 6000-3.4) 13.D.1
- Puntatore 10
- Rappresentazione in PASCAL 6000-3.4 13.B.2
- Record 7.
- Scalare 5.A
- Set 8.
- Sottocampo 5.B
- Standard (PASCAL 6000-3.4) 13.9.3
- Vettore 6.

Valori di verità 2.A
Variabili
- Di controllo 4.C.3
- Globali 11.A
- Locali 11.A
- Parte dichiarazione 3.E

Vocabolario 1.

RAPPORTO

1. Introduzione

Il linguaggio PASCAL è stato sviluppato puntando a due obiettivi principali: anzitutto rendere disponibile un linguaggio adatto all'insegnamento della programmazione intesa come una disciplina sistematica, basata su dei concetti base riflessi dal linguaggio in modo chiaro e naturale; indi sviluppare delle implementazioni che siano efficienti ed affidabili sugli elaboratori attualmente disponibili.

Il desiderio di un nuovo linguaggio per l'insegnamento della programmazione mi è nato dall'insoddisfazione dei riguardi dei principali linguaggi attualmente usati: linguaggi i cui costrutti e le cui caratteristiche, troppo spesso, non possono essere spiegati in modo logico e convincente e, troppo spesso, si oppongono ad un ragionamento sistematico. Oltre a questa insoddisfazione c'è la mia convinzione che il linguaggio con cui si insegna allo studente ad esprimere le sue idee influenza profondamente sia la sistematicità che l'inventiva del suo pensare, e che il disordine insito in questi linguaggi si riflette direttamente nello stile di programmazione degli studenti.

Ci sono naturalmente moltissime ragioni per andar cauti nell'introdurre un nuovo linguaggio e ci sono senza dubbio delle giustificazioni nel porre obiezioni contro l'insegnamento della programmazione in un linguaggio non ancora ampiamente diffuso ed accettato perlomeno in un'ottica commerciale a breve termine. Tuttavia, la scelta di un linguaggio per l'insegnamento, basata sull'ampiezza della sua disponibilità ed accettazione, unito al fatto che il linguaggio più diffusamente insegnato diviene poi quello più usato dà la ricetta più sicura per il ristagno in un campo di così profonda influenza pedagogica. Io ritengo valga la pena di fare uno sforzo per rompere questo circolo vizioso.

Naturalmente un nuovo linguaggio non dovrebbe essere sviluppato solo per amor di novità; i linguaggi esistenti potrebbero essere usati come base per lo sviluppo, a patto che non vadano contro i criteri menzionati e non impediscano una struttura sistematica. ALGOL 60 è stato infatti usato come base per Pascal perché risponde alle esigenze di programmazione in una maniera molto migliore che qualsiasi altro linguaggio standard. Perciò i principi di strutturazione e la forma delle espressioni sono copiati da Algol 60. Non si è tuttavia ritenuto corretto adottare Algol 60 come un sottoinsieme di Pascal; certi principi di costruzione, in particolare quelle delle dichiarazioni, sarebbero stati incompatibili con quelli che avrebbero permesso un'adatta e naturale rappresentazione delle caratteristiche addizionali di Pascal.

Le maggiori estensioni rispetto ad Algol 60 riguardano le possibilità di strutturazione dei dati: infatti la causa principale del relativamente piccolo campo di applicazioni di Algol 60 era considerata la sua rigidità nella strutturazione dei dati. L'introduzione di strutture record e flusso dovrebbe rendere possibile la soluzione di problemi commerciali con Pascal, o, come minimo l'uso per la soluzione di tali problemi in un corso di programmazione.

2. Sommario del linguaggio

Un algoritmo o programma è costituito da due parti essenziali: una descrizione delle *azioni* che devono essere fatte, ed una descrizione dei *dati* che devono essere elaborati da queste azioni. Le azioni sono descritte dalle *istruzioni* ed i dati dalle *dichiarazioni* e *definizioni*.

I dati sono rappresentati da valori di *variabili*. Ogni variabile che ricorre in un'istruzione deve essere introdotta da una *dichiarazione* di variabile che associa un identificatore ed un tipo di dato con quella variabile. Un *tipo di dato* definisce essenzialmente l'insieme di valori che possono essere assunti da quella variabile. Un tipo di dato, in Pascal, può essere descritto direttamente nella dichiarazione della variabile, oppure vi si può far riferimento con un identificatore di tipo; in questo secondo caso l'identificatore deve essere esplicitamente descritto da una *definizione di tipo*.

I tipi di dato base sono i tipi *scalari*. La loro definizione indica un insieme ordinato di valori: introduce cioé un identificatore per ogni valore dell'insieme. Oltre ai tipi scalari definibili, esistono quattro *tipi base standard*: *Booleano intero*, *carattere* e *reale*. Eccezion fatta per il tipo booleano, i loro valori non sono denotati da identificatori, ma rispettivamente da numeri e "citazioni" che sono sintatticamente distinti dagli identificatori. L'insieme di valori di tipo carattere sono i caratteri disponibili nella particolare installazione.

Un tipo - indicando il minore ed il maggior valore del campo può essere definito come un sottocampo di un tipo scalare.

I *tipi strutturati* sono definiti descrivendo i tipi dei loro componenti ed indicando il *metodo di strutturazione*. I vari metodi di strutturazione differiscono nel meccanismo di selezione dei componenti di una variabile di tipo strutturato. In Pascal sono disponibili quattro metodi base di strutturazione: strutturazione a vettori, record, set e flussi.

In una *struttura vettore* tutti i componenti sono dello stesso tipo. Ogni componente è scelto da un selettore di vettore, o *indice calcolabile*, il cui tipo (che deve essere scalare) è indicato nella definizione del tipo del vettore: solitamente è un tipo scalare definito dal programmatore oppure un sottocampo del tipo intero. Dato un valore dell'indice, un selettore di vettore dà un valore del tipo

componente. Il tempo necessario per una selezione non dipende dal valore del selettore (indice). I vettori sono perciò delle *strutture ad accesso casuale.*

In una *struttura record*, i componenti (chiamati *campi*) non sono necessariamente dello stesso tipo. Un selettore di record non è un valore calcolabile ma è un identificatore che in modo univoco denota il componente scelto: questo fa si che il tipo del componente scelto sia chiaro del testo del programma (senza l'esecuzione). Questi identificatori di componenti sono dichiarati nella definizione del record. Come per i vettori il tempo necessario per accedere ad un componente non dipende dal selettoe, e perciò anche il record è una struttura ad accessi casuale.

Un record può essere dichiarato costituito da parecchie *varianti.* Questo significa che variabili diverse, sebbene dichiarate dello stesso tipo, possono assumere strutture per certi aspetti diverse. Possono differire e per il numero e per il tipo dei componenti. Un campo, comune a tutte le varianti, chiamato *campo discriminatore* indica la variante associata al valore corrente di una variabile record. Solitamente, la parte comune a tutte le varianti è costituita, oltre che dal campo discriminatore, da parecchi componenti.

Una *struttura flusso* è una sequenza di componenti dello stesso tipo. Nella sequenza è definito un ordinamento naturale dei componenti e, ad ogni richiesta, si può accedere direttamente ad un solo componente. Si può accedere agli altri componenti avanzando sequenzialmente lungo il flusso. Un flusso è generato appendendo i suoi componenti in coda e quindi la definizione del tipo flusso non determina il numero dei componenti.

Le variabili dichiarate in dichiarazionu esplicite sono dette *statiche*. La dichiarazione associa alla variabile un identificatore che è usato per far riferimento alla variabile. È comunque possibile generare delle variabili anche con istruzioni eseguibili. Questa generazione, *dinamica*, dà un cosidetto *puntatore* (sostituzione di un identificatore esplicito) che nel seguito serve per riferirsi alla variabile. Questo puntatore può essere assegnato ad altre variabili, cioé a variabili di tipo puntatore. Ogni puntatore può assumere solo valori che puntano a variabili dello stesso tipo T e vien detto *limitato* a questo tipo T. Può, tuttavia, assumere anche il valore **nil**, nel qual caso non punta ad alcuna variabile. Poiché le variabili puntatore possono essere anche componenti di variabili strutturate, il loro uso permette una completa rappresentazione di grafi finiti.

L'istruzione fondamentale è l'*istruzione di assegnamento*: permette che un nuovo valore calcolato sia assegnato ad una variabile (o a componenti di una variabile). Il valore è ottenuto calcolando una *espressione*. Le espressioni sono costituite da variabili, costanti, insiemi, operatori e funzioni applicati alle quantità denotate e generanti i nuovi valori. Le variabili, le costanti e le

funzioni sono dichiarate nel programma o sono standard. Pascal definisce un insieme finito di operatori, ciascuno dei quali può essere considerato come la descrizione della "mappatura" dei tipi degli operandi sul tipo del risultato. L'insieme degli operatori è suddiviso nei seguenti gruppi:

1. *operatori aritmetici* di addizione, sottrazione, segno, inversione, moltiplicazione, divisione e calcolo del resto.
2. *operatori booleani* di negazione, unione (or) e congiunzione (and).
3. *operatori su insiemi* di unione, intersezione e differenza di insiemi.
4. *operatori relazionali* di eguaglianza, ineguaglianza, ordinamento, appartenenza ed inclusione di insiemi. I risultati di operazioni relazionali sono di tipo *booleano*.

L'*istruzione di chiamata di procedura* provoca l'esecuzioe della procedura chiamata (si veda più avanti). Gli assegnamenti e le chiamate di procedure sono i componenti o mattoni base delle *istruzioni strutturate*, che specificano un'esecuzione sequenziale, selettiva o ripetuta dei loro componenti. L'esecuzione sequenziale di istruzioni è specificata dall'*istruzione composta*, ed infine quella ripetuta dalle istruzioni di **repeat, while** e **for**. L'istruzione **if** serve per rendere l'esecuzione di un'istruzione dipendente dal valore di un'espressione booleana, mentre l'istruzione **case** permette la selezione tra parecchie istruzioni in base al valore di un selettore. L'istruzione **for** è usata quando il numero delle iterazioni è conosciuto in anticipo; negli altri casi sono usati **repeat** e **while**.

Ad un'istruzione può essere dato un nome (identificatore) che serve per farvi riferimento. L'istruzione è chiamata *procedura* e la sua dichiarazione *dichiarazione di procedura*. La dichiarazione può inoltre contenere un insieme di dichiarazioni di variabili, definizione di tipi e dichiarazione di altre procedure. Si può far riferimento a variabili, tipi e procedure, dichiarati all'interno della procedura (e perciò detti *locali*), solo all'interno della procedura stessa. I loro identificatori hanno significato solo nel testo del programma che costituisce le dichiarazioni di procedura e che è chiamato l'*ambiente* di questi identificatori.

Poiché le procedure possono essere dichiarate locali ad altre procedure, gli ambienti possono nidificare. Le entità dichiarate nel programma principale, vale a dire non locali ad alcuna procedura, sono dette *globali*. Una procedura ha un numero fisso di parametri, ciascuno dei quali è denotato all'interno della procedura da un identificatore chiamato *parametro formale*.
All'attivazione di una procedura, deve essere indicata una quantità effettiva per ogni parametro a cui si faccia riferimento da dentro la procedura stessa con un parametro formale. Questa quantità è chiamata *parametro effetivo*. Ci sono quattro tipi di parametri: parametri valori, variabile, funzione e procedura. Nel primo caso il parametro effettivo è un'espressione che è valutata una sola volta. Il parametro formale rappresenta una variabile locale alla quale è

assegnato il risultato di questa valutazione prima dell'esecuzione della procedura (funzione). Nel caso di un parametro variabile, il parametro effettivo è una variabile ed il parametro formale sta in luogo di questa variabile. Gli eventuali indici sono calcolati prima dell'esecuzione della procedura (o funzione). Nel caso di parametri procedura e funzione, il parametro effettivo è un identificatore di procedura o funzione.

Le *funzioni* sono dichiarate analogamente alle procedure. La sola differenza sta nel fatto che una funzione dà un risultato che può essere solo di tipo scalare o puntatore e che va specificato nella dichiarazione della funzione. Le funzioni possono perciò essere usate come elementi delle espressioni. Per eliminare gli effetti collaterali, nelle dichiarazioni delle funzioni andrebbero evitati gli assegnamenti a variabili non locali.

3. Notazioni, terminologia, vocabolario

In accordo con la tradizionale forma di BACKUS-NAUR, i costrutti sono denotati da termini italiani racchiusi tra le parentesi angolari <e>. Questi termini descrivono anche la natura o il significato del costrutto, e sono poi usati nella descrizione del significato. Le possibili ripetizioni di un costrutto sono indicate racchiudendolo tra le metaparentesi {e}. Il simbolo <vuoto> sta ad indicare una sequenza senza simboli.

Il vocabolario base di Pascal è costituito dai simboli base classificabili in: lettere, cifre e simboli speciali.

<lettera> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l |
m | n | o | p | q | r | s | t | u | v | w | x | y | z |

<cifra> ::= | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<simbolo speciale> ::=
+ | - | * | / | = | <> | < | > | <= | >= | (|) | [|] | { | } | := | . | , | ; | : | ' | ↑
div	**mod**	**nil**	**in**	**or**	**and**	**not**	**if**	**then**	**else**	**case**	**of**
repeat	**until**	**while**	**do**	**for**	**to**	**downto**	**begin**	**end**
with	**goto**	**const**	**var**	**type**	**record**	**set**	**file**
function	**procedure**	**label**	**packed**	**program**

Il costrutto

{ <qualsiasi sequenza di simboli non contenente"}" > }

può essere inserito tra due qualsiasi identificatori, numeri (cfr. 4), o simboli

speciali: è chiamato *commento* e può essere tolto dal testo del programma senza che se se sia alterato il significato. I simboli {e} non compaiono con altro significato nel linguaggio; quando sono usato nelle descrizioni sintattiche sono dei metasimboli come | e ::=. La coppia di simboli (* e *) è usata come sinonimo per {e}.

4. Identificatori, numeri, stringhe

Gli identificatori servono per denotare costanti, tipi, variabili, procedure e funzioni. La loro associazione deve essere unica nel campo di validità: all'interno, cioé, della procedura o funzione in cui sono dichiarati (cfr. 10 e 11).

```
<identificatore> ::= <lettera> <lettera o cifra>
<lettera o cifra> ::= <lettera> | <cifra>
```

Per i numeri, i quali sono costanti del tipo *intero* o *reale*, è usata la consueta notazione decimale (si veda 6.1.2). La lettera E, quando precede il fattore di scala, è letta come "volte, 10 elevato a".

```
<sequenza di cifre> ::= <cifra> | <cifra>
<intero senza segno> ::= <sequenza di cifre>
<reale senza segno> ::= <intero senza segno> . <sequenza di cifre> |
                        <intero senza segno> . <sequenza di cifre> E
                        <fattore di scala> | <intero senza segno> E
                        <fattore di scala>
<numero senza segno> ::= <intero senza segno> |
                         <reale senza segno>
<fattore di scala> ::= <intero senza segno> | <segno>
                       <intero senza segno>
<segno> ::= +|-
```

Esempi:

1 100 0.1 5E-3 87.35E+8

Le sequenze di caratteri racchiuse tra apici sono chiamate *stringhe*. Le stringhe costituite da un solo carattere solo le costanti del tipo standard *char* (si veda 6.1.2). Le stringhe costituite da n (n > 1) caratteri sono costanti del tipo (si veda 6.2.1)

packed array [1..n] **of** char

Nota: se la stringa deve contenere un apice, questo apice va scritto due volte.

```
<stringa> ::= ' <carattere> { <carattere> } '
```

Esempi:

'A' ';' ''''
'PASCAL' 'ecco una stampa'

5. Definizioni delle costanti

Una definizione di costante introduce un identificatore come sinonimo per la costante.

<identificatore costante> ::= <identificatore>
<costante> ::= <numero senza segno> | <segno>
<numero senza segno> | <identificatore costante>
| <segno> <identificatore costante> | <stringa>
<definizione costante> ::= <identificatore> = <costante>

6. Definizione dei tipi dei dati

Il tipo determina l'insieme di valori che le variabili di quel tipo possono assumere ed associa un identificatore con il tipo.

<tipo> ::= <tipo semplice> | <tipo strutturato> |
<tipo puntatore>
<definizione tipo> ::= <identificatore> = <tipo>

6.1 Tipi semplici

<tipo semplice> ::= <tipo scalare> | <tipo sottocampo> |
<identificatore tipo>
<tipo semplice> ::= <identificatore>

6.1.1 Tipi scalari

Un tipo scalare definisce un insieme ordinato di valori enumerando gli identificatori che denotano questi valori.

<tipo scalare> ::= (<identificatore {, <identificatore> })

Esempi (Rosso, arancio, giallo, verde, blu)
(fiori, quadri, cuori, picche)
(lunedì, martedì, mercoledì, giovedì, venerdì, sabato, domenica)

Le funzioni applicabili a tutti i tipi scalari (eccettuati i reali) sono:
succ: calcola il valore seguente (nell'enumerazione)
pred: calcola il valore precedente (nell'enumerazione)

6.1.2 Tipi standard

In Pascal i tipi standard sono i seguenti:

integer (intero)	I valori sono un sottoinsieme dell'insieme dei numeri interi. Questo sottoinsieme dipende dall'implementazione ed i suoi valori sono gli interi definiti in 4.
real (reale)	I suoi valori sono un sottoinsieme dei numeri reali il quale dipende dall'implementazione. I valori sono denotati dai numeri reali (si veda 4).
Boolean (Booleani)	I suoi valori sono i valori, della tabella della verità, denotati dagli identificatori true e false.
char (carattere)	I suoi valori sono un insieme di caratteri determinati della particolare implementazione. Sono denotati dai caratteri stessi racchiusi tra apici.

6.1.3 Tipi sottocampo

Un tipo può essere definito come un sottocampo di un tipo scalare indicando il più piccolo ed il più grande valore del sottocampo. La prima costante specifica il limite inferiore, e non deve essere maggiore del limite superiore.

<tipo sottocampo> ::= <costante> .. <costante>

Esempi:

1..100
-10..+10
lunedi..martedi

6.2 Tipi strutturati

Un tipo strutturato è caratterizzato dal tipo dei suoi componenti e dal metodo di strutturazione. Inoltre la definizione di un tipo strutturato può contenere un'indicazione della rappresentazione preferita dei dati. Se una definizione ha

come prefisso il simbolo **packed**, non ha effetto alcuno sul significato del programma (per una restrizione si veda 9.1.2); è però un suggerimento al compilatore per economizzare memoria anche al prezzo di una certa perdita di efficienza nell'accesso; ciò può, inoltre, espandere il codice necessario per accedere ai componenti della struttura.

<Tipo strutturato> ::= <tipo strutturato disimpaccato> |
packed <tipo strutturato disimpaccato>

<tipo strutturato disimpaccato> ::= <tipo vettore> | <tipo record> |
<tipo set> | <tipo flusso>

6.2.1 Tipi vettore

Un tipo vettore è una struttura costituita da un numero fisso di componenti, tutti dello stesso tipo: *tipo componente*. Gli elementi del vettore sono individuati da indici, valori appartenenti al cosidetto *tipo indice*. La definizione del tipo vettore specifica sia il tipo componente che l'indice.

<tipo vettore> ::= **array** [<tipo indice> {, <tipo indice> }] **of**
<tipo componente>
<tipo indice> ::= <tipo semplice>
<tipo componente> ::= <tipo>

Se sono specificati n tipi indici, il tipo vettore è detto *n-dimensionale*, ed un componente è designato con n indici.

Esempi:

array [1..100] **of** real
array [1..10, 1..20] **of** 0..99
array [Boolean] **of** colori

6.2.2 Tipi record

Un tipo record è una struttura costituita da un numero fisso di componenti, anche di tipo diverso. La definizione del tipo record specifica, per ogni componente, chiamato *campo*, il suo tipo ed un identificatore che lo denota. l'ambiente di questi *identificatori di campo* è la definizione stessa del record; i suoi componenti sono accessibili tramite un designatore di campo (cfr. 7.2) che fa riferimento ad una variabile record.

Un tipo record può avere parecchie *varianti*, nel qual caso un campo può essere designato come *campo discriminatore*. Il valore del campo discriminatore indica quale è la variante che la variabile record ha assunto in quel dato momento. Ogni variante è identificata da un'etichetta di selezione che è una costante del tipo del campo discriminatore.

```
<tipo record> ::= record <elenco campi> end
<elenco campi> ::= <parte fissa> | <parte fissa>; <parte variante> |
                   <parte variante>
<parte fissa> ::= <sezione record> {; <sezione record> }
<sezione record> ::= <identificatore di campo>
                     {, <identificatore di campo> }:
                     <tipo> | <vuoto>
<parte variante> ::= case <campo discriminatore> <identificatore tipo>
                     of <variante> {; <variante> }
<variante> ::= <elenco etichette selezione> : ( <elenco campi> ) |
               <vuoto>
<elenco etichette selezione> ::= <etichette selezione>
                                 {, <etichetta selezione> }
<etichetta selezione> ::= <costante>
<campo discriminatore> ::= <identificatore> | <vuoto>
```

Esempi:

```
record giorno : 1..31;
       mese  : 1..12;
       anno  : integer

end

record nome, cognome : alfa;
       eta : 0..99;
       sposato: Boolean

end

record x, y : real;
       area : real;
    case s : forma of
    triangolo: (lato : real;
                iclinazione, angolo1, angolo2: angolo);
    rettangolo: (lato1, lato2 : real;
                 sbieco, angolo3 : angolo);
    cerchio: (diametro : real)
end
```

6.2.3 Tipi set

Un tipo set definisce la gamma di valori che è l'insieme potenza dei suoi cosidetti *tipi base*. I tipi base non devono essere tipi strutturati. Gli operatori

applicabili a tutti i tipi insieme sono:

+ unione
- differenza di insiemi
* intersezione
in appartenenza

La differenza di insiemi x-y è definita come l'insieme di tutti gli elementi di x che non apartengono a y.

<tipo set> ::= **set of** <tipo base>
<tipo base> ::= <tipo semplice>

5.2.4 Tipi flusso

Una definizione di tipo flusso specifica una struttura costituita da una sequenza di componenti tutti dello stesso tipo. Il numero dei componenti - la *lunghezza* del flusso - non è fissata nella definizione del tipo. Un flusso con 0 componenti è detto *vuoto*.

<tipo flusso> ::= **file of** <tipo>

I flussi con componenti di tipo carattere sono chiamati flussi-testo; sono un caso particolare: infatti la gamma dei valori dei componenti deve essere considerata ampliata con un simbolo che denota la fine di una linea. Questo simbolo individua, nei flussi-testo, delle sottostrutture: le linee. Il tipo *text* è un tipo standard dichiarato come:

type <text> = **file of** <char>

6.3 Tipi puntatore

Le variabili dichiarate in un programma (si veda 7.) sono accessibili per mezzo dei loro identificatori; esistono durante tutta l'esecuzione della procedura (ambiente) rispetto alla quale la variabile è locale: vengono perciò dette *statiche* (o allocate staticamente). Le variabili possono comunque essere generate dinamicamente: senza cioé correlazione alcuna con la struttura del programma. Queste variabili *dinamiche* sono generate dalla procedura standard **new** (si veda 10.1.2); dal momento che non compaiono in una dichiarazione esplicita di variabili, non vi si può far riferimento con un nome. Vi si accede, invece, per mezzo di un valore cosidetto *puntatore* che è fornito alla generazione della variabile dinamica. Un tipo puntatore consiste perciò in un insieme illimitato di valori che puntano ad elementi dello stesso tipo. Sui puntatori sono definite solo le operazioni di assegnamento e verifica di eguaglianza. Il valore puntatore **nil** appartiene ad ogni tipo di puntatore e punta a "nessun elemento".

<tipo puntatore> ::= ↑ <identificatore tipo>

Esempi di definizione tipo:

```
colore    = (rosso, giallo, verde, blu)
sesso     = (maschio, femmina)
text      = file of char
forma     = (triangolo, rettangolo, cerchio)
scheda    = array [1..80] of char
alfa      = packed array [1..10] of char
complesso = record re, im : real end
persona   = record
                nome, cognome : alfa;
                eta : integer;
                sposato : boolean;
                padre, figlio, consorte : ↑ persona
            case s ; sesso of
                maschio: (militesente, militassolto: boolean);
                femmina: (incinta: boolean;
                          taglia : array [1..3] of integer)
            end
```

7. Dichiarazioni e denotazioni di variabili

Le dichiarazioni di variabili sono costituite da un elenco di identificatori denotanti le nuove variabili, seguiti dal loro tipo.

<dichiarazione variabile> ::= <identificatore>
{, <identificatore> } : <tipo>

Ogni dichiarazione di una variabile flusso f con componenti di tipo T implica la dichiarazione di una cosidetta *variabile buffer* di tipo T. Questa variabile buffer è denotata da f ↑ e serve, nella generazione, per aggiungere (in coda) componenti al flusso e nella ricerca per accedere al flusso (si veda 7.2.3 e 10.1.1).

Esempi:

```
x, y, z: real
u, v,: complesso
i, j,: integer
k: 0..9
p, q: Boolean
operatore: (più, meno, per)
a: array [0..63] of real
b: array [colore, boolean] of complessi
c: colore
f: file of char
hue1, hue2: set of colore
p1, p2: ↑ persone
```

Una denotazione di variabile può designare una variabile completa od un componente di una variabile, o una variabile individuata da un puntatore (si veda 6.3). Le variabili che si incontreranno negli esempi dei capitoli seguenti si supporranno dichiarate come indicato sopra.

<variabile> ::= <variabile completa> | <variabile componente> |
<variabile indirizzata>

7.1 Variabili complete

Una variabile completa è denotata dai suoi identificatori.

<variabile completa> ::= <identificatore variabile>
<identificatore variabile> ::= <identificatore>

7.2 Variabili componenti

Un componente di una variabile è denotato dalla variabile seguita da un selettore che specifica il componente. La forma del selettore dipende dal tipo di strutturazione della variabile.

<variabile componente> ::= <variabile con indice> |
<designatore di campo> |
<buffer flusso>

7.2.1 Variabili con indice

Un componente di una variabile a n-dimensioni è denotato dalla variabile seguito da n espressioni degli indici.

<variabile con indice> ::= <variabile vettore> [<espressione>
{, <espressione> }]
<variabile vettore> ::= <vettore>

I tipi delle espressioni degli indici devono essere gli stessi dei tipi dichiarati nella definizione del vettore

Esempi:

a [12]
a [i + j]
b [red, true]

7.2.2 Designatori di campo

Un componente di una variabile record è denotato dalla variabile record seguita dall'identificatore di campo del componente.

<designatore di campo> ::= <variabile record> .
<identificatore di campo>
<variabile record> ::= <variabile>
<identificatore di campo> ::= <identificatore>

Esempi:

u.re
b [red, true] . im
p2 ↑ . size

7.2.3 Buffers flusso

In ogni istante è accessibile direttamente solo un componente determinato dalla posizione corrente del flusso (posizione di lettura/scrittura). Questo componente è chiamato componente corrente ed è rappresentato dalla *variabile buffer* del flusso.

<buffer flusso> ::= <variabile flusso> ↑
<variabile flusso> ::= <variabile>

7.3 Variabili indirizzate

<variabile indirizzata> ::= <variabile puntatore> ↑
<variabile puntatore> ::= <variabile>

Se p è una variabile puntatore ad un tipo T, p denota quella variabile ed il suo valore, mentre p ↑ denota la variabile di tipo T indirizzata da p.

Esempi:

p1 ↑ .padre
p1 ↑ .consorte ↑ .figlio

8. Espressioni

Le espressioni sono costrutti denotanti regole di calcolo per ottenere valori di variabili e generare nuovi valori con l'applicazione di operatori. Le espressioni sono costituite da operatori ed operandi, cioé variabili, costanti e funzioni.

Le regole di composizione specificano delle *precedenze* in accordo con 4 classi di operatori. L'operatore **not** ha la priorità maggiore, seguito dai cosidetti operatori moltiplicativi, dagli operatori additivi ed infine dagli operatori relazionali. Una sequenza di operatori della stessa precedenza sono eseguiti da sinistra a destra. Le regole di precedenza sono riflesse dalla sintassi seguente:

<costante senza segno> ::= <numero senza segno> | <stringa> |
| <identificatore costante> | **nil**
<fattore> ::= <variabile> | <costante senza segno> |
<designatore di funzione> | <set> |
(<espressione>) | **not** fattore
<set> ::= <elenco elementi>
<elenco elementi> ::= <elemento> {, <elemento> } | <vuoto>
<elemento> ::= <espressione> | <espressione> .. <espressione>
<termine> ::= <fattore> | <termine> <operatore multiplicativo>
<fattore>
<espressione semplice> ::= <termine> | <espressione semplice>
<operatore additivo> <termine> |
<segno> <termine>
<espressione> ::= <espressione semplice> |espressione semplice>
<operatore relazionale> <espressione semplice>

Le espressioni che appartengono ad un insieme devono essere tutte dello stesso tipo: il tipo base dell'insieme. [] denota l'insieme vuoto, e [x..y] denota l'insieme di tutti i valori nell'intervallo x...y.

Esempi:

Fattori:

x
15
(x+y+2)
sin (x+y)
[rosso, c, verde]
[1, 5, 10..99, 23]
not p

Termini:

x * y
i / (1-i)
p **or** q
(x < = y) **and** (x < z)

Espressioni semplici:

x+y
-x
hue 1 + hue 2
i * j + 1

Espressioni:

x = 1.5
p < = q
(i < j) = (j < k)
c **in** hue 1

8.1 Operatori

Se entrambi gli operandi degli operatori aritmetici di addizione, sottrazione e moltiplicazione sono di tipo intero (o sottocampo), il risultato è di tipo intero. Se uno degli operandi è reale anche il risultato è reale.

8.1.1. L'operatore not

L'operatore **not** denota la negazione del suo operando Booleano.

8.1.2. Operatori moltiplicativi

<operatori moltiplicativi> ::= * | / | **div** | | **mod** | **and**

Operatore	Operazione	Tipo operandi	Tipo risultato
*	moltiplicazione intersezione di insiemi	reali, interi ogni tipo insieme T	reale, intero T
/	divisione	reali, interi	reale
div	divisione con troncamento	interi	intero
mod	modulo	intero	intero
and	"and" logico	Booleani	Booleano

8.1.3. Operatori additivi

<operatori additivi> ::= + | -| **or**

Operatore	Operazione	Tipo operandi	Tipo risultati
+	addizione unione di insiemi	interi, reali qualsiasi tipo	intero, reale T
-	sottrazione differ. di insiemi	interi, reali qualsiasi tipo insieme T	intero, reale T
or	"or" logico	Booleani	Booleano

Quando sono usati con un solo operando, - indica inversione di segno, e +

indica l'operazione identità.

8.1.4 Operatori relazionali

<operatori relazionali> ::= =|<>|<|<=|>=|>| **in**

Operatore	Tipo operandi	Risultato
= <> < > <= >=	qualsiasi scalare o sottocampo	Booleano
in	qualsiasi scalare o sottocampo ed il suo tipo insieme	Booleano

Si noti che tutti i tipi scalari definiscono un insieme *ordinato* di valori. Gli operatori <>, <=, >= significano rispettivamente: diverso, minore o uguale, maggiore o uguale.
Gli operatori < = e > = possono anche essere usati per confrontare valori di tipo set e quindi denotano inclusioni.
Se p e q sono espressioni Booleane, p=q denota la loro equivalenza, e p < =q denota che q è implicato da p. (Si noti che false < true)

Gli operatori relazionali =, <> < <=, >>= possono anche essere usati per confrontare vettori (impaccati) con componenti di tipo char (stringhe), e denotano l'ordinamento alfabetico in accordo con la sequenza dell'insieme di caratteri sottostanti.

8.2 Designatori funzionali

Un designatore funzionale specifica l'attivazione di una funzione. Consiste in un identificatore che designa la funzione ed in un elenco di parametri effettivi. I parametri sono variabili, espressioni, procedure e funzioni, e sostituiscono i corrispondenti parametri formali (cfr. 9.1.2, 10, 11).

<designatore di funzione> ::= <identificatore di funzione> |
<identificatore di funzione>
(<parametro effettivo>
{, <parametro effettivo> })
<identificatore di funzione> ::= <identificatore>

Esempi: Somma (a, 100)
MCD (147, k)
sin (x+y)
eof (f)
ord (f ↑)

9. Istruzioni

Le istruzioni sono *eseguibili* e denotano delle azioni. Possono essere precedute da un'etichetta a cui si può far riferimento con un'istruzione di goto.

```
<istruzione> ::= <istruzione senza etichetta> | <etichetta> :
                 <istruzione senza etichetta>
<istruzione senza etichetta> ::= <istruzione semplice> |
                                 | <istruzione strutturata>
<etichetta> ::= <intero senza segno>
```

9.1 Istruzioni semplici

Un'istruzione semplice è un'istruzione che ha le sue parti costituite da altre istruzioni. L'istruzione vuota è costituita da nessun simbolo e non denota azione alcuna.

```
<istruzione semplice> ::= <istruzione di assegnamento> |
                          <istruzione procedura> |
                          <istruzione goto> | <istruzione vuota>
```

9.1.1 Istruzioni di assegnamento

L'istruzione di assegnamento serve per sostituire il valore corrente di una variabile con un nuovo valore specificato come espressione.

```
<istruzione di assegnamento> ::= <variabile> := <espressione> |
                                 <identificatore di funzione> :=
                                 <espressione>
```

La variabile (o funzione) e l'espressione devono essere dello stesso tipo; sono permesse le seguenti eccezioni:

1. il tipo della variabile è reale ed il tipo dell'espressione intero o sottocampo
2. il tipo dell'espressione è un sottocampo del tipo della variabile, o viceversa.

Esempi:

```
x := y+z
p := (i <= i) and (i < 100)
i := sqr (k) - (i * j)
hue 1 := [blu, succ (c)]
```

9.1.2 Istruzioni procedura

Un'istruzione procedura serve per eseguire la procedura denotata dall'identificatore di procedura. L'istruzione può contenere un elenco di *parametri effettivi* che sostituiscono i corrispondenti *parametri formali* definiti nella dichiarazione di procedura (si veda 10).
La corrispondenza tra i parametri è posizionale; esistono 4 tipi di parametri: parametri valore, variabile, procedura (il parametro effettivo è un identificatore di procedura), e funzione (il parametro effettivo è un identificatore di funzione).

Nel caso di *parametro valore* il parametro effettivo deve essere un'espressione (della quale una variabile è un caso particolare). Il corrispondente parametro formale rappresenta una variabile locale della procedura chiamata ed il valore corrente dell'espressione è assegnato, all'inizio, a questa variabile.
Nel caso di *parametro variabile* il parametro effettivo deve essere una variabile effettiva durante l'intera esecuzione della procedura. Se questa variabile è un elemento di un vettore il suo indice è calcolato al momento della chiamata della procedura. Un parametro variabile deve essere usato tutte le volte che il parametro rappresenta un risultato.

I componenti di una struttura impaccata non devono apparire come parametri variabile effettivi.

```
<istruzione procedura> ::= <identificatore di procedura> |
                           <identificatore di procedura>
                           ( <parametro effettivo>
                           {, <parametro effettivo> } )
<identificatore di procedura> ::= <identificatore>
<parametro effettivo> ::= <espressione> | <variabile> |
                          <identificatore di procedura> |
                          <identificatore di funzione>
```

Esempi: prossimo
Scambia (a, n, m)
Bisect (fct, - 1.0, + 1.0, x)

9.1.3 Istruzione goto

L'istruzione di goto indica che l'elaborazione deve proseguire a partire dall'istruzione che ha per etichetta l'etichetta associata al *goto* stesso.

<istruzione goto> ::= **goto** <etichetta>

Le etichette devono rispettare le seguenti restrizioni:

1. L'ambiente di una etichetta è la procedura entro cui è stata definita: non è perciò possibile saltare in una procedura.

2. Ogni etichetta deve essere dichiarata nell'intestazione della procedura nella quale l'etichetta individua l'istruzione.

9.2 Istruzioni strutturate

Le istruzioni strutturate sono costrutti composti da altre istruzioni che devono essere eseguiti o in sequenza (istruzione composta), o in modo condizionato (istruzioni condizionali) o in modo ripetitivo (istruzioni iterative).

<istruzione strutturata> ::= <istruzione composta> |
<istruzione condizionale> |
<istruzione iterativa> |
<istruzione with>

9.2.1 Istruzioni composte

L'istruzione composta specifica che le sue istruzioni componenti devono essere nello stesso ordine con cui sono scritte. I simboli **begin** e **end** fungono da parentesi per l'istruzione.

<istruzione composta> ::= **begin** <istruzione> {; <istruzione> } **end**

Esempio: **begin** z := x; x := y; y := z **end**

9.2.2 Istruzioni condizionali

Un'istruzione condizionale sceglie per l'esecuzione una sola delle sue istruzioni componenti.

<istruzioni condizionali> ::= <istruzione if> | <istruzione case>

9.2.2.1 Istruzioni if

L'istruzione if specifica che un'istruzione va eseguita solo se è vera una certa condizione (espressione booleana). Se è falsa o non viene eseguita alcuna istruzione o viene eseguita quella che segue il simbolo **case**.

<istruzione if> ::= **if** <espressione> **then** <istruzione> | **if**
<espressione> **then** <istruzione> **else** <istruzione>

L'espressione tra **if** e **then** deve essere Booleana.

Nota:
L'ambiguità sintattica che nasce dal costrutto:

```
if <espressione 1> then if <espressione 2> then <istruzione 1>
                        else <istruzione 2>
```

è risolta interpretando il costrutto come equivalente a:

```
if <espressione 1> then begin if <espressione 2> then <istruzione 1>
else <istruzione 2>
end
```

Esempi:

```
if x< 1.5 then z := x + y else z := 1.5
if p1 <> nil then p1 := p1 ↑ . padre
```

9.2.2.2 Istruzioni case

L'istruzione di case è costituita da un'espressione (il selettore) e da un elenco di istruzioni, ognuna delle quali è marcata con una costante dello stesso tipo del selettore. Essa specifica che deve essere eseguita l'istruzione che ha l'etichetta uguale al valore corrente del selettore.

```
<istruzione case> ::= case <espressione> of
                      <elenco elementi case>
                      {; <elenco elementi case> } end
<elenco elementi case> ::= <elenco etichette case> :
                           <istruzione> | <vuoto>
<elenco etichette case> ::= <etichette case> {, <etichette case> }
```

Esempi:

```
case operatore of
più: x := x+y;
meno: x := x-y;
per: x := x*y
end
```

```
case i of
1: x := sin (x);
2: x := cos (x);
3: x := exp (x);
4: x := ln (x);
end
```

9.2.3 Istruzioni di Iterazione

Le istruzioni iterative specificano che certe istruzioni devono essere eseguite ripetutamente. Se il numero di ripetizioni è conosciuto in anticipo - prima che

le ripetizioni partano - il costrutto più appropriato è **for**; negli altri casi vanno usati **while** o **repeat**.

<istruzione iterativa> ::= <istruzione while>
| <istruzione repeat> | <istruzione for>

9.2.3.1 Istruzione while

<istruzione while> ::= **while** <espressione> **do** <istruzione>

L'espressione che controlla la ripetizione deve essere di tipo booleano; l'istruzione è ripetuta fino a che l'espressione diviene falsa. Se l'espressione ha un valore false fin dall'inizio, l'istruzione non viene mai eseguita.

L'istruzione:

```
while B do S
```

è equivalente a:

```
if B then
    begin S;
        while B do S
    end
```

Esempi:

```
while a [i] <> x do i := i + 1
while i > 0 do
begin if odd (i) then z := z*x;
        i := i div 2;
        x := sqr (x)

end

while not eof (f) do
begin P (f↑); get (f)
end
```

9.2.3.2 Istruzioni Repeat

<istruzione repeat> ::= **repeat** <istruzione> {; <istruzione> }
until <espressione>

L'espressione che controlla la ripetizione deve essre di tipo booleano. La

frequenza di istruzioni fra i simboli **repeat** e **until** è ripetutamente eseguita (ed almeno una volta) finché l'espressione diviene vera.

L'istruzione:
repeat S **until** B

è equivalente a

```
begin S;
    if not B then
        repeat S until B
end
```

Esempi:

```
repeat k := i mod j;
       i := j;
       j := k
until  j = 0
repeat P (f↑); get (f)
until eof (f)
```

9.2.3.3 Istruzioni for

L'istruzione for indica che un'istruzione deve essere ripetutamente eseguita mentre una progressione di valori è assegnata ad una variabile chiamata *variabile di controllo* dell'istruzione for.

```
<istruzione for> ::= for <variabile di controllo> := <elenco for>
                     do <istruzione>
<elenco for> ::= <valore iniziale> to <valore finale> |
                 <valore iniziale> downto <valore finale>
<variabile di controllo> ::= identificatore>
<valore iniziale> ::= <espressione>
<valore finale> ::= <espressione>
```

Le variabili di controllo, il valore iniziale ed il valore finale devono essere dello stesso tipo scalare (o sottocampo), e non devono essere cambiati dall'istruzione che viene ripetuta. Non possono essere di tipo reale.

Un'istruzione for del tipo

for v := e 1 **to** e 2 **do** S

è equivalente alla sequenza di istruzioni

```
v := e 1; s; v := succ (v); S; ...; v := e 2; S
```

ed un'istruzione for del tipo

```
for v := e1 downto e2 do S
```

è equivalente a

```
v := e1; S; v := pred (s); S; ...; v := e2; S
```

Esempi:

```
for i := 2 to 63 do if a [i] > max then max := a [i]

for i := 1 to n do
for j := 1 to n do
begin x := 0;
    for k := 1 to n do x := x+A [i,k]* B[k, j];
    C [i, j] := x
end
for c := rosso to blu do Q (c)
```

9.2.4 Istruzioni with

```
<istruzione with> ::= with <elenco variabili record> do <istruzione>
<elenco variabili record> ::= <variabile record>
                              {, <variabile record>}
```

Nell'istruzione che segue il *do* (istruzione componente), i campi delle variabili record specificati da with possono essere denotati solo dal loro identificatore di campo: non devono cioé essere preceduti dalla denotazione di tutta la variabile record. In effetti il with apre l'ambiente contenente gli identificatori di campo della variabile record specificata di modo che gli identificatori di campo possono essere usati come identificatori di variabile.

Esempio:

```
with data do
if mese := 12 then
    begin mese := 1; anno := anno + 1
    end
else mese := mese + 1
```

è equivalente a:

```
if data.mese = 12 then
    begin data.mese := 1; data.anno := data.anno + 1
    end
else data.mese := data.mese + 1
```

Nell'istruzione associata a with non possono essere fatti assegnamenti a nessun elemento dell'elenco delle variabili record; sono invece possibili assengnamenti ai componenti di queste variabili.

10. Dichiarazioni di procedure

Le dichiarazioni di procedure servono per definire parti di programmi ed associarvi degli identificatori cosicché possono essere attivate dalle istruzioni procedura.

```
<dichiarazione procedura> ::= <intestazione procedura> <blocco>
<blocco> ::= <parte dichiarazione etichette>
             <parte definizione costanti> <parte definizine tipo>
             <parte dichiarazione variabili>
             <parte dichiarazione procedure e funzioni>
             <parte istruzioni>
```

L'*intestazione di procedura* specifica l'identificatore che dà il nome alla procedura e gli eventuali identificatori dei parametri formali. I parametri possono essere parametri valore, variabile, procedura o funzione (si veda anche 9.1.2). Le procedure e funzioni che sono usate come parametri per altre procedure o funzioni devono avere solo parametri valore.

```
<intestazione di procedura> ::= procedure <identificatore>; |
                                procedure <identificatore>
                                ( <sezione parametri formali>
                                {; <sezione parametri formali> } );
<sezione parametri formali> ::= <gruppo parametri> | var
                                <gruppo parametri> |
                                function <gruppo parametri> |
                                procedure <identificatore>
                                {, <identificatore> }
<gruppo parametri> ::= <identificatore> {, <identificatore> }:
                       <identificatore tipo>
```

Un gruppo parametri che non sia preceduto da una parola di specificazione ha come elementi dei parametri valore.

La *parte dichiarazione etichette* specifica tutte le etichette che individuano un'istruzione nella parte istruzioni.

<parte dichiarazione etichette> ::= <vuoto> | **label** <etichetta>
{, <etichetta> };

La *parte definizione costanti* contiene la definizione di tutti i sinonimi delle costanti locali alla procedura.

<parte definizione costanti> ::= <vuoto> | **const**
<definizione costanti>
{; <definizione costanti> };

La *parte definizione tipi* contiene tutte le definizioni di tipo che sono locali alla procedura.

<parte definizione tipi> ::= <vuoto> | **type** <definizione tipo>
{; <definizione tipo> };

La *parte dichiarazione variabili* contiene la dichiarazione di tutte le variabili locali alla procedura.

<parte dichiarazione variabili> ::= <vuoto> | **var**
<dichiarazione variabile>
{; <dichiarazione variabile> };

La *parte dichiarazione procedura e funzioni* contiene le dichiarazioni di tutte le procedure e funzioni locali alla procedura.

<parte dichiarazione procedura e funzioni> ::=
{ <dichiarazione procedura o funzione> ;}
<dichiarazione procedura o funzione> ::=
<dichiarazione procedura> | <dichiarazione funzione>

La *parte istruzioni* specifica le azioni che devono essere eseguite quando, tramite una chiamata, è attivata la procedura.

<parte istruzioni> ::= <instruzione composta>

Tutti gli identificatori introdotti nella parte dei parametri formali (parte di definizione costanti e tipo, parte di dichiarazione variabili, procedure o funzioni) sono *locali* alla dichiarazione di procedura che vien detta *ambiente* (scope) di questi identificatori. Nel caso di variabili locali, i loro valori sono indefiniti all'inizio della parte istruzioni.

L'uso dell'identificatore di procedura, in una chiamata all'interno della sua dichiarazione implica l'esecuzione recursiva della procedura stessa.

Esempi di dichiarazione di procedure:

```
procedure leggintero (var f : text; var x : integer);
var i, j : integer;
begin while f ↑ = ' ' do get (f); i := 0;
    while f ↑ in '0' .. '9' do
        begin j := ord (f ↑)- ord ('0');
              i := 10* i+j;
              get (f)
        end;
    x := i
end

procedure bisec (function f : real; a, b : real; var z:real);
var m : real;
begin {si assume che f (a) < 0 e f (b) > 0{
    while abs (a-b) > 1E-10*abs (a) do
    begin m := (a+b) / 2.0;
        if f (m) < 0 then a := m else b := m
    end;
    z := m
end
procedure MCD (m, n integer; var x, y, z : integer);
var a1, a2, c, d, q, r : integer;  {m >= 0, n > 0}
begin {Massimo comun divisore x tra m e n; algoritmo di Euclide esteso}
    a1 := 0; aç := 1; b1 := 1; b2 := 0;
    c := m; d := n;
    while d <> o do
    begin {a1 * m+b1 * n = d, a2 * m+b2 * n = c,
        mcd (c, d) = mcd (m, n) }
        q := c div d; r := c mod d;
        a2 := a2-q*a1; b2 := b2-q*b1;
        c := d; d := r;
        r := a1; a1 := a2; a2 := r;
        r := b1; b1 := b2; b2 := r
    end

    x := c; y := a2; z := b2
    {x = mcd (mn) = y*m + z*n}
end
```

10.1 Procedure standard

Tutte le implementazioni di Pascal devono avere le procedure standard già dichiarate; possono, naturalmente, dichiararne anche altre. Poiché queste procedure, come tutti gli standard, sono dichiarate in un ambiente che circonda il programma, non sorgono conflitti con dichiarazioni che ridefiniscono gli stessi identificatori dentro il programma stesso. Le procedure standard sono elencate e spiegate nel sottoparagrafo seguente.

10.1.1 *Procedure per il trattamento dei flussi*

put (f) — appende il valore della variabile buffer f ↑ al flusso f. L'effetto è definito solo se prima dell'esecuzione il predicato eof (f) è vero; eof (f) rimane vero, ed il valore di f diventa indefinito.

get (f) — posiziona la testina (logica) di lettura/scrittura sul prossimo componente del flusso (nuova posizione corrente). ed assegna il valore di questo componente alla variabile buffer f ↑. Se non esiste un prossimo componente eof (f) diventa vero ed il valore di f ↑ non è definito. L'effetto di get (f) è definito solo se eof (f) = false prima dell'esecuzione (si vesa 11.1.2)

reset (f) — riporta la posizione corrente del flusso all'inizio ed assegna alla variabile f ↑ il valore del primo elemento di f; eof (f) diviene falso se f non è vuoto; nel caso contrario f ↑ non è definito ed eof (f) rimane vero.

rewrite (f) — scarta il valore corrente di f di modo che può essere generato un nuovo flusso. eof (f) diviene vero.

Per quanto riguarda le procedure read, write, readln, writlen e page si veda il capitolo 12.

10.1.2 *Procedure di allocazione dinamica*

new (p) — alloca una nuova variabile v ed assegna alla variabile puntatore p il puntatore a v. Se v è di tipo record con variante, la forma

new (p, t1, ..., tn)
può essere usata per allocare una variabile della variante con valore del campo discriminatore t1, ... tn. I valori del campo discriminatore, che devono essere elencati in modo contiguo e nell'ordine di dichiarazione, non devono essere cambiati durante l'esecuzione.

dispose (p) indica che la memoria occupata dalla variante p non è più necessaria. Se per allocare la variabile è stata usata la seconda forma di new, allora

dispose (p, t1, ... tn)
con valori del campo discriminatore *identici* deve essere usata per indicare che la memoria occupata da questa variante non è più necessaria.

10.1.3 *Procedure di trasferimento dati*

Siano le variabili a e z dichiarate da

```
a : array [m..n] of T
z : packed array [u..v] of T
```

in cui n-m >= v-u. L'istruzione pack (a, i, z) significa

```
for j := u to v do z [j] := a [j-u+i]
```

e l'istruzione unpack (z, a, i) significa

```
for j := u to v do a [j-u+i] := z [j]
```

in cui j denota una variabile ausiliaria che non ricorre in altre parti del programma.

11. Dichiarazioni di funzioni

Le dichiarazioni di funzione servono per definire parti di programmi che computano un valore scalare o puntatore. Le funzioni sono attivate dalla valutazione di un designatore di funzione (8.2) che è un elemento di un'espressione.

```
<dichiarazione di funzione>::= <intestazione di funzione>
                               <blocco>
```

L'intestazione specifica il nome dell'identificatore che individua la funzione, i parametri formali ed il tipo della stessa.

```
<intestazione di funzione> ::= function <identificatore>
                               : <tipo risultato>; | function
                               <identificatore>
                               ( <sezione parametro formale>
                               {; <sezione parametro formale> } )
                               : <tipo del risultato>;
<tipo risultato> ::= <identificatore tipo>
```

Il tipo di una funzione deve essere scalare, sottocampo e puntatore. All'interno della dichiarazione di una funzione ci deve essere almeno un'istruzione di

assegnamento che assegna un valore all'identificatore di funzione: l'assegnamento determina il risultato della funzione. Il fatto che ci sia l'identificatore di funzione in una designazione di funzione all'interno della propria dichiarazione implica un'esecuzione recursiva della funzione stessa.

Esempi:

```
function Sqrt (x:real): real;

var x0, x1 : real;
begin x1 := x; {x > 1, metodo di Newton}
    repeat x0 := x1; x1 := (x0 + x/x0) * 0,5
    until abs (x1 - x0) < eps * x1;
    Sqrt := x0
end

function Mass (a:vettore; n:integer): real;
var x : real; i : integer;
begin x := a 1;
    for i := 2 to n do
    begin {x = mass (a [1],..., a [i-1] ) }
        if x < a [i] then x := a [i]
    end
    {x = max (a [1] a [n])}
    Mass := x
end

function MCD (m, n:integer): integer;
begin if n = 0 then MCD := m
      else MCD := MCD (n, m mod n)
end

function Power (x:real; y:integer): real; {y>= 0}
var w, z : real; i : integer;
begin w := x; z := 1; i := y;
    while i > 0 do
    begin {z* (w**i)= x**y}
        if odd (i) then z := z*w;
        i  := i div 2
        w := sqr (w)
    end;
    {z = x**y}
    Power := z
end
```

11.1 Funzioni standard

Ogni implementazione di Pascal deve avere già dichiarate le funzioni standard; può naturalmente dichiararne altre (si veda anche 10.1)

Le funzioni standard sono elencate e spiegate, nei sottoparagrafi che seguono.

11.1.1 *Funzioni artimetiche*

abs (x) — calcola il valore assoluto di x può essere sia del tipo reale che intero ed il tipo del risultato è il medesimo di x.

sqr (x) — calcola x**2. X può essere sia di tipo reale che intero ed il risultato è dello stesso tipo di x.

sin (x)
cos (x)
exp (x)
ln (x)
sqrt (x)
arctan (x)

x può essere sia di tipo intero che reale ma il risultato è di tipo reale.

11.1.2 *Predicati*

odd (x) — x deve essere di tipo intero ed il risultato è vero se x è dispari e falso nell'altro caso.

eof (f) — eof (f) indica se il flusso f è nello stato di fine flusso (end-of-file).

eoln — indica la fine di una linea in un flusso-testo (si veda il cap. 12)

11.1.3 *Funzioni di trasformazione*

trunc (x) — il valore reale x è troncato della parte decimale

round (x) — l'argomento x è arrotondato all'intero più vicino

ord (x) — x deve essere di tipo scalare (Booleano e carattere inclusi), ed il risultato (di tipo intero) è il numero ordinale del valore x nell'insieme definito dal tipo di x.

chr (x) — x deve essere di tipo intero, è il risultato (di tipo carattere) è il carattere (se esiste) il cui numero ordinale è x.

11.1.4 *Altre funzioni standard*

succ (x) x può essere scalare o sottocampo ed il risultato - se esiste - è il successore di x.

pred (x) x può essere scalare o sottocampo ed il risultato - se esiste - è il predecessore di x.

12. Input e Output

La base degli input/output leggibili sono i flussi-testo (6.2.4) che vengono passati come parametri (13) ad un programma PASCAL e nel suo ambiente rappresentano certi dispositivi di input/output come terminale, lettore di schede o stampante.
Per facilitare la gestione dei flussi-testo, oltre alle procedure standard *get* e *put* sono introdotte le procedure *read, write, readln, writeln.* I flussi-testo a cui queste procedure standard si applicano non devono necessariamente rappresentare dei dispositivi di Input/Output, ma possono anche essere dei flussi locali. Le nuove procedure sono usate con una sintassi non standard per quanto riguarda l'elenco dei paramtri; tra le altre cose è permesso un numero variabile di parametri. Inoltre i parametri non devono essere di tipo carattere: nel caso in cui siano di qualche altro tipo il trasferimento dei dati è associato ad una convenzione implicita. Se il primo parametro è una variabile flusso, allora questo è il flusso che va letto o scritto. I valori per difetto nel caso di lettura e scrittura sono i flussi standard *input* e *output*. Questo due flussi sono predichiarati come:

var input, output : text

I flussi-testo rappresentano un caso speciale in quanto i testi sono sottostrutturati in linee per mezzo di marche (si veda 6.2.4). Se, leggendo un flusso-testo f, la posizione nel flusso raggiunge una marca di fine linea (si trova cioé dopo l'ultimo carattere della linea), il valore della variabile buffer f ↑ diventa uno spazio, e la funzione standard *eoln (f)* (*e*nd *o*f *l*inee - fine linea) dà il valore vero. Avanzando di un'altra posizione, la variabile buffer f ↑ diviene il primo carattere della prossima linea ed eoln (f) dà il valore falso (a meno che la prossima linea sia costituita da 0 caratteri). Le marche di linea, non essendo elementi di tipo carattere, possono essere generate solo dalla procedura *writeln.*

12.1 Procedure read

Si assuma che f denoti un flusso-testo e v1...vn delle variabili di tipo carattere, intero (o sottocampo di intero), o reale; allora per la procedura read valgonô le seguenti regole:

1. read (v1, ... ,vn) è equivalente a read (input, v1 ... , vn)

2. read (f, v1, ... , vn) è equivalente a read (f, v1); ...; read (f, vn)

3. se v è una variabile di tipo **char**, read (f, v) e equivalente v := f↑; get (f)

4. se v è una variabile di tipo intero (o sottocampo di intero) o reale, allora (f, v) implica la lettura da f di una sequenza di caratteri che formano un numero in accordo con la sintassi di PASCAL (si veda 4) e l'assegnamento di questo numero a v. Gli spazi che precedono i caratteri e le marche di linea non sono considerati.

La procedura read può essere usata per leggere da un flusso f che non sia un testo; read (f, x) è in questo caso equivalente a x := f ↑; get (f).

12.2 Procedura readln

1. readln (v1, ... , vn) è equivalente a readln (input, v1, ... , vn)

2. readln (f, v1, ... , vn) è equivalente a read (f, v1, ... , vn); readln (f)

3. readln (f) è equivalente a

```
while not eoln (f) do get (f);
get (f)
```

Readln è usata per leggere e poi passare all'inizio della prossima linea.

12.3 Procedura write

Si supponga che f denoti un flusso-testo; p1, ... , pn i cosidetti parametri di scrittura; e un'espressione; m ed n espressioni di tipo intero. Per la procedura *write* valgono allora le seguenti regole:

1. write (p1, ... , pn) è equivalente a write (output, p1, ... , pn)

2. write (f, p1, ... , pn) è equivalente a write (f, p1,); ...; write (f, pn)

3. I parametri di scrittura p possono avere le seguenti forme:

e:m e:m:n e

deve e rappresenta il valore che deve essere scritto nel flusso f ed m ed n sono i parametri di "lunghezza campo". Se il valore e, che può essere un numero, un carattere, un valore booleano o una stringa richiede meno di m caratteri per la sua rappresentazione allora si ha in uscita un numero adeguato di spazi, di modo che sono scritti proprio m caratteri. Se è omesso, si assume un valore per difetto che dipende dall'implementazione. La forma con il parametro n è applicato solo se e è di tipo reale (si veda la regola 6).

4. Se è di tipo **char**, si ha che
   write (f, e:m) è equivalente a:
   f ↑ := ' '; put (f); (ripetuta m-1 volte)
   f ↑ := e; put (f)
   NOTA: Il valore per difetto di m in questo caso è L

5. Se e è di tipo intero (o sottocampo di intero), allora la rappresentazione decimale del numero e sarà scritta nel flusso f preceduta da un appropriato numero di spazi come specificato da m.

6. Se è di tipo **real**, si scriverà nel flusso f un appropriato numero di spazi come specificato da m e la rappresentazione decimale di e. Se non c'é il parametro n (regola 3), si sceglierà un rappresentazione in virgola mobile costituita da un coefficiente e da un fattore di scala. Se c'é n si avrà una rappresentazione in virgola fissa con n cifra dopo il punto (virgola).

7. Se e è di tipo **Boolean**, dopo averla fatta precedere dal numero di spazi determinato da m, si scrivono le parole TRUE e FALSE.

8. Se e è un vettore impaccato di caratteri, la stringa e è scritta nel flusso f preceduta dal numero di spazi determinato da m.

La procedura write può anche essere usata per scrivere in un flusso f che non è un flusso-testo. In questo caso write (f, x) è equivalente a f ↑ := x; put (f).

12.4 Procedura writeln

1. writeln (p1, ... , pn) è equivalente a writeln (output, p1, ... ,pn)

2. writeln (f, p1, ... , pn) è equivalente a write (f, p1, ... ,pn); writeln (f)

3. writeln (f) appende una marca di fine linea al flusso f (cfr. 6.2.4)

page (f) fa si che si salti in testa ad una nuova pagina, quando è stampato il flusso-testo f.

13. Programmi

Un programma PASCAL, eccettuata l'intestazione, ha la stessa forma di una dichiarazione di procedura.

<programma> ::= <intestazione programma> <blocco>
<intestazione programma> ::=
    program identificatore (parametri di programma);
<parametri di programma> ::= <identificatore> {, <identificatore> }

L'identificatore che segue il simbolo **program** è il nome nel programma; all'interno del programma non ha altri significati. I parametri di programma denotano entità che esistono fuori del programma, e che permettono al programma di comunicare con l'ambiente in cui è inserito. Queste entità (di solito flussi) sono dette *esterne*, e devono essere dichiarate nel blocco che costituisce il programma come qualsiasi variabile locale.
i due flussi standard *input* e *output* non devono essere dichiarati (cfr. 12), ma, se sono usati, devono essere elencati come parametri nell'intestazione del programma. Le istruzioni di inizializzazione reset (input) e rewrite (output) sono generate automaticamente e non devono essere specificate dal programmatore.

Esempi:

```
program copia (f, g);
var f, g : file of real;
begin reset (f); rewrite (g);
    while not eof (f); do
        begin g ↑ := f ↑; put (g); get (f)
        end
end

program coptesto (input, output);
var ch : char;
begin
    while not eof (input); do
    begin
        while not eoln (input); do
            begin read (ch); write (ch)
            end;
        readln; writeln
    end
end.
```

14. Standards implementativi per la portabilità

Uno dei motivi principali per lo sviluppo di PASCAL era la necessità di avere un linguaggio potente e flessibile che potesse essere implementato su molti elaboratori in modo ragionevolmente efficiente. Le sue caratteristiche sono state definite senza far riferimento ad alcuna macchina per facilitare lo scambio dei programmi. In questo capitolo si riporta un insieme di restrizioni progettate come guida per implementatori e programmatori che ritengono che i loro programmi possono essere usati su elaboratori diversi. Lo scopo di questi standards è di aumentare la possibilità che implementazioni diverse siano compatibili e che i programmi siano trasferibili da un'installazione ad un'altra.

1. Gli identificatori che denotano oggetti distinti devono differire per i primi 8 caratteri.

2. Le etichette devono essere costituite da almeno 4 cifre.

3. L'implementazione può fissare un limite alle dimensioni del tipo base sul quale si possa definire un set. (Come conseguenza, è ragionevole rappresentare gli insiemi con una stringa di bits).

4. Il primo carattere di ogni linea di un flusso stampabile può essere interpretato come un carattere di controllo per la stampante con i seguenti significati:

spazio	:	spaziatura singola
'0'	:	spaziatura doppia
'1'	:	stampa in testa alla prossima pagina
'+'	:	nessuna interlinea (sovrascritturata)

Le rappresentazioni di PASCAL (nell'insieme dei caratteri disponibili) dovrebbe seguire le seguenti regole:

5. Le parole simbolo - come **begin**, **end**, ecc. - sono scritte con una sequenza di lettera (senza caratteri di fuga inframmezzati). Non possono essere usate come identificatori.

6. Gli spazi, i fine linea ed i commenti sono considerati separatori. Tra due simboli PASCAL consecutivi può esserci un numero qualsiasi di separatori con, però, la seguente restrizione: all'interno degli identificatori, numeri e parole simbolo non possono esserci separatori.

7. Deve esserci almeno un separatore tra ogni coppia di identificatori, numeri o parole simbolo consecutivi.

15. Indice analitico

Assegnamento	9.1.1
Blocco	10.
Buffer flusso	7.2.3
Campo discriminatore	6.2.2
Case	9.2.2.2
Cifra	3.
Costante	5.
Costante senza segno	8.
Definizione costante	5.
Definizione tipo	6.
Designatore di campo	7.2.2
Designatore di funzione	8.2
Dichiarazione di funzione	11.
Dichiarazione procedura	10.
Dichiarazione procedura o funzione	10.
Dichiarazione variabile	7.
Elemento	8.
Elenco campi	6.2.2
Elenco elementi	8.
Elenco elementi case	9.2.2.2
Elenco etichette case	9.2.2.2 e 6.2.2
Elenco for	9.2.3.3
Elenco variabili record	9.2.4
Espressione	8.
Espressione semplice	8.
Etichetta	9.
Etichetta selezione	6.2.2
Fattore	8.
Fattore di scala	4.
Gruppo parametri	10.
Identificatore	4.
Identificatore campo	7.2.2
Identificatore costante	5.
Identificatore di funzione	8.2
Identificatore di procedura	9.1.2
Identificatore tipo	6.1
Identificatore variabile	7.1
Intero senza segno	4.
Intestazione di funzione	11.
Intestazione di procedura	10.

Intestazione programma 13.
Istruzione 9.
Istruzione composta 9.2.1
Istruzione condizionale 9.2.2
Istruzione for 9.2.3.3
Istruzione goto 9.1.3
Istruzione if 9.2.2.1
Istruzione iterativa 9.2.3
Istruzione procedura 9.1.2
Istruzione repeat 9.2.3.2
Istruzione semplice 9.1
Istruzione senza etichetta 9.
Istruzione strutturata 9.2
Isruzione vuota 9.1
Istruzione while 9.2.3.1
Istruzione with 9.2.4
Lettera 3.
Lettera o cifra 4.
Numero senza segno 4.
Operatori additivi 8.1.3
Operatori moltiplicativi 8.1.2
Operatori relazionali 8.1.4
Parametri effettivi 9.1.2
Parametri programma 13.
Parte definizione costanti 10.
Parte definizione tipi 10.
Parte dichiarazione etichette 10.
Parte dichiarazione procedure e funzioni 10.
Parte dichiarazioni variabili 10.
Parte fissa 6.2.2
Parte istruzioni 10.
Parte variante 6.2.2
Programma 13.
Reale senza segno 4.
Segno 4.
Sequenza di cifre 4.
Set 8.
Sezione parametri formale 10.
Sezione record 6.2.2
Simbolo speciale 3.
Stringa 4.
Termine 8.
Tipo 6.
Tipo base 6.2.3

Tipo componente 6.2.1
Tipo flusso 6.2.4
Tipo indice 6.2.1
Tipo puntatore 6.3
Tipo record 6.2.2
Tipo risultato 11.
Tipo scalare 6.1.1
Tipo semplice 6.1
Tipo set 6.2.3
Tipo sottocampo 6.1.3
Tipo strutturato 6.2
Tipo strutturato disimpaccato 6.2
Tipo vettore 6.2.1
Valore finale 9.2.3.3
Valore iniziale 9.2.3.3
Variabile 7.
Variabile completa 7.1
Variabile componente 7.2
Variabile con indice 7.2.1
Variabile di controllo 9.2.3.3
Variabile flusso 7.2.3
Variabile indirizzata 7.3
Variabile puntatore 7.3
Variabile record 7.2.2
Variabile vettore 7.2.1
Variante 6.2.2

L. 10.000

Cod. 500-P

42

pascal

MANUALE E STANDARD DEL LINGUAGGIO

Kathleen Jensen
Niklaus Wirth

GRUPPO EDITORIALE JACKSON